# Preventing lost messages in event-driven programming

## UCLA CSD Technical Report TR060001
## January 25, 2006, revised July 3, 2006

Jeffrey Fischer Rupak Majumdar Todd Millstein

Department of Computer Science, University of California, Los Angeles
{fischer,rupak,todd}@cs.ucla.edu

**Abstract.** The *event-driven* programming style is pervasive as an efficient method for interacting with the environment. Unfortunately, the event-driven style severely complicates program maintenance and understanding, as it requires each logical flow of control to be fragmented across multiple independent callbacks.
We propose a backward-compatible extension to Java, called TaskJava, which supports lightweight, interleaved computations without foregoing standard control mechanisms like procedures and exceptions. At the same time, TaskJava programs can be automatically translated into efficient event-based code.
This technical report presents, in detail, a formalization of TaskJava. We formalize the operational semantics, typing rules, and modular compilation strategy for a core sublanguage. We then prove a number of properties of this core language, including type soundness, observational equivalence of translated programs, and freedom from certain classes of bugs.

## 1 Introduction

A wide variety of applications, from high-performance servers to enterprise applications to GUIs to embedded systems, rely on an *event-based* programming style [18]. Event-driven programming implements a stylized programming idiom where programs use non-blocking I/O operations, and the programmer breaks the computation into fine-grained *callbacks* (or *event handlers*) that are each associated with the completion of an I/O call (or *event*). This approach permits the interleaving of many simultaneous logical tasks with minimal overhead, under the control of an application-level cooperative scheduler. Each callback executes some useful work and then either schedules further callbacks, contingent upon later events, or invokes a *continuation*, which resumes the control flow of its logical caller. The event-driven style has been demonstrated to achieve high throughput in server applications [19, 24], resource-constrained embedded devices [12], and business applications [4].

Threads represent an alternative programming model commonly used for many of these applications. However, although threads are useful and convenient

for many situations, threads have disadvantages as well, including the potential for race conditions and deadlocks, as well as high memory consumption [23]. Within the systems research community, there is currently no agreement that one approach is better than the other [18, 19, 22, 23, 2]. In addition, in some contexts, threads either cannot be used at all (such as within an operating system kernel) or can only be used in conjunction with events (such as thread-pooled servers for Java Servlets [4]). Thus, we believe that events are here to stay and an important target for programming language improvements.

Unfortunately, programming with events comes at a cost: event-driven programs are extremely difficult to understand and maintain. Each logical unit of work must be manually broken into multiple callbacks scattered throughout the program text. This manual code decomposition is often in conflict with higher-level program structuring. For example, calls do not return directly to their callers, so it is difficult to make use of procedural abstraction as well as a structured exception mechanism. The event-driven style also loses many of the benefits of object-oriented programming. Rather than modeling a concept from the application domain as a class, one must artificially break the functionality into multiple callback classes, which conform to a generic interface for use by the scheduler. Aside from being tedious and unnatural, this style impedes the use of inheritance to implement related domain concepts, making code reuse difficult. Overall, the event-driven style makes it difficult for both programmers and programming tools to reason about the runtime behavior of an application, and can lead to difficult to track errors.

We make two contributions in this paper. First, we introduce *tasks* as a new programming model and associated language construct for interleaved computation that enables the programmer to apply ordinary higher-level program structuring constructs such as exceptions and inheritance while maintaining the flexibility and efficiency of event-driven programming. Second, we provide a formalization of event driven programming using TaskJava and the compilation of TaskJava, and use the formalization to prove type safety as well as demonstrate that certain common classes of errors in event-driven programs cannot occur in TaskJava.

**TaskJava.** For the first contribution, we present the design of a backward-compatible extension to Java called TaskJava that incorporates tasks. Similar to a Java thread, a TaskJava task encapsulates an independent unit of work in a method called `run`. In this way, the logical control flow of each unit of work is preserved. At the same time, tasks can be automatically implemented by the compiler in an event-driven style, thereby obtaining the low-overhead and high-throughput advantages of events. To achieve this goal, tasks employ non-blocking I/O libraries, as in event-driven systems. Rather than registering a callback with each I/O call, however, programmers use a special `wait` primitive that we provide. Conceptually, a `wait` causes the current task to block until one of a specified set of events occurs. Our compiler uses a task's `wait` calls to automatically convert the `run` method into an equivalent set of callbacks. More specifically, the `run` method is broken by the compiler at each `wait` call into

two separate methods, one that executes up to the `wait` call (which is replaced with an event registration) and one to continue after a waited-for event is signaled. This implementation strategy is achieved by employing a restricted form of *continuation-passing style* (CPS), a well-studied compiler transformation that is popular for functional programming languages [1, 3].

In addition to being called by a task's `run` method, `wait` may be called by methods on non-task classes that are declared as *asynchronous*. Asynchronous methods are written and invoked like standard methods, but they are translated by the compiler using CPS, in the same manner as a task's `run` method. Asynchronous methods enable procedural abstraction and layering in the presence of non-blocking calls.

By enabling language-level abstractions, TaskJava can prevent two significant classes of errors that arise in event-driven programs: the *lost continuation problem* and the *lost exception problem*. The lost continuation problem occurs when a callback has an execution path in which the callback's continuation is neither invoked nor passed along to the next callback in the event chain. A lost continuation causes the intended sequential behavior of the program to be broken, often producing errors that are difficult to trace to their source. Because TaskJava methods return to their callers as usual, the lost continuation problem is avoided.

The lost exception problem occurs when an error condition produced by a callback is not properly handled by the subsequent continuation, potentially causing the program to crash or continue executing in undefined ways. This problem arises because the continuation-passing nature of the event-driven style subverts the usage of language-level exception-handling mechanisms. Instead, exceptional conditions are typically encoded as special return values that are passed on to the continuation. This approach loses the static guarantee that all exceptions are handled, as the continuation can easily fail to check for all possible error scenarios. In contrast, TaskJava allows the programmer to use the same exception mechanisms as in Java, and can use a similar analysis to check that a TaskJava program does not raise any uncaught checked exceptions (of course, as in Java, programs can still raise runtime exceptions that may not be caught).

**Compiler implementation and case study.** We have implemented the TaskJava compiler in the Polyglot compiler framework [17]. To gain experience with tasks and to concretely illustrate their benefits, we extended Fizmez [5], an open source web server, to use interleaved computation. We implemented two versions: one using hand-coded, explicit continuation passing and one using TaskJava. In the TaskJava implementation, the control flow of the original program is preserved. In the hand-coded implementation, the control flow is implicit and broken across many classes. In our performance tests, the TaskJava implementation performed with a maximum 16% addition in latency, and 44% addition in througput compared with the hand-tuned version. Since the TaskJava implementation has not been optimized, we believe that the performance penalties can be further reduced, while retaining the language-level benefits of tasks.

**CoreTaskJava.** For the second contribution, we formalize the TaskJava language and its implementation strategy. To do this, we define two core languages that extend Featherweight Java (FJ) [14]:

- CoreTaskJava (CTJ), which extends FJ with tasks and asynchronous methods.
- EventJava (EJ), which extends FJ to support explicit events and event registration.

CTJ formalizes TaskJava and EJ formalizes low level event-driven programming. We provide the operational semantics and static typing rules for both languages and a translation relation from CTJ to EJ, which serves to formalize the continuation-passing transformation performed by the TaskJava compiler. We use these formal systems to prove several key properties about CTJ programs:

- *Type soundness*: This property ensures that well-typed CTJ programs don't get "stuck" (i.e., incur a run-time type error), and it is proven in the standard "progress and preservation" style [25].
- *Translation correctness*: This property ensures that a CTJ program and its EJ translation are observationally equivalent. The property is proven by showing that the execution traces of the two programs are bisimulation equivalent.

In particular, as a corollary of the type soundness theorem, we obtain the no lost exception property: a well-typed CTJ program never throws an uncaught checked exception. We also formalize the no lost continuation property in the core languages. Since methods in CTJ never directly generate continuations, we can prove the lost continuation property for CTJ. Further, as a corollary of translation correctness, we can conclude that the event-driven implementation of CTJ programs cannot have lost continuations. On the other hand, general EJ programs may have lost continuations. The proof uses the bisimulation equivalence to generate dynamic traces in the original CTJ program and its translation. The full proofs for these theorems are provided in a separate technical report [9].

**Organization.** The rest of the paper is organized as follows. In Section 2, we give an informal overview of tasks, demonstrate through an example what TaskJava programs look like, and contrast with event-driven programs. In Section 3, we formalize the TaskJava model and compilation algorithm in a core calculus. In Section 4, we then state and prove the central formal properties of CTJ programs: subject reduction, progress, and soundness. Finally, in Section 5, we formalize a translation strategy and prove the observational equivalence between CTJ programs and their EventJava translations. In Section 6, we describe our web server case study. We survey related work in Section 8 and conclude in Section 9.

## 2 Programming with Tasks

This section informally introduces the TaskJava language extensions and demonstrates their benefits for implementing event-based applications. We use a running example consisting of a simple web server written in an event-driven style. We will contrast a standard event-driven implementation with an implementation taking advantage of the asynchronous programming features provided by TaskJava.

The server processes simple one-line HTTP requests for files, whose contents are sent to the client. To keep the example short, we omit some details that would be needed in a realistic implementation (character set translation, multi-line requests, and the processing of long files a block at a time). We assume that each connection between a client and the server, called a *channel* in Java terminology, supports the following events:

- `READ_RDY_EVT`: This event signals that there is incoming data available to be read from a channel.
- `WRITE_RDY_EVT`: This event signals that the associated channel is ready for writing.
- `ACPT_RDY_EVT`: This event signals an accept request is available on the associated channel.
- `ERR_EVT`: This event signals that an error has occurred on the channel.

Event-driven systems employ a user-level scheduler, which accepts requests for event notifications and schedules the execution of these notifications. The implementation of these schedulers varies widely, based on the abstractions of the underlying I/O library and on differing design choices.

In our example, `static final` fields are used as an enumeration to identify event types of interest. A set of these *event identifiers* is passed to a scheduler to indicate interest in an event. The scheduler returns an *event object* containing the details of an event which occurred (event type, channel, error information, etc.). The event objects are instances of an `Event` class, which is defined as follows:

```
public class Event {
  // Enumeration of event identifiers
  // provided by the scheduler.
  public static final int READ_RDY_EVT = 1;
  public static final int WRITE_RDY_EVT = 2;
  public static final int ACPT_RDY_EVT = 3;
  public static final int ERR_EVT = 4;

  // type () returns which kind of event occurred.
  public int type() { ... }
  // getError() returns the event's error,
  // or null if no error occurred.
  public IOException getError() { ... }
}
```

```
01 public interface Callback {
02   void run(Object retVal);
03 }
04 public interface Continuation extends Callback {
05   void error(Exception e);
06 }
07 public class EventIO {
08   public static class Reader {
09     ...
10     public void readLine(Continuation cont) {
11       Scheduler.register(ch, Event.READ_RDY_EVT,
12                          Event.ERR_EVT,
13                          new ReadCb(this, cont));
14     }
15     private static class ReadCb implements Callback {
16       ...
17       public void run(Object retVal) {
18         Event event = (Event)retVal;
19         if (event.type() == Event.READ_RDY_EVT) {
20           rdr.ch.read(rdr.cbuf); // do the actual read
21           String line = scanChars();
22           if (line!=null) cont.run(line);
23           else Scheduler.register(rdr.ch,
24                                   Event.READ_RDY_EVT,
25                                   Event.ERR_EVT, this);
26         }
27         else { // ERR_EVT
28           assert event.type() == Event.ERR_EVT;
29           cont.error(event.getError());
30         }
31       }
32     }
33   }
34   public static void write(CharChannel ch,
35                            CharBuffer data,
36                            Continuation cont) {
37     ...
38   }
39   public static void accept(CharChannel ch,
40                             Continuation cont) {
41      ...
42   }
43   ...
44 }
```

**Fig. 1.** An event-driven nonblocking I/O library

Other event definitions are possible. As described in section 6.2, TaskJava is scheduler-agnostic and can be easily adapted to different schedulers.

### 2.1 An Event-driven Web Server

**Event-driven I/O library** We first describe a simple event-driven I/O library implemented on top of Java's nonblocking I/O primitives (figure 1). This library interacts with an event scheduler (not shown) by making a call to `Scheduler.register`, which associates a set of events (specified using event identifiers) for a given I/O channel with a provided callback. When any one of these events occurs, an event object is passed to the `run` method of the callback (the callback's interface is defined in lines 1 - 3). The registration of an event set is one-time: after one of the events has occurred, the entire set of events associated with the callback is removed from the scheduler. Of course, a callback may re-register itself or a different callback with the scheduler.

Unlike most libraries, an event-driven library cannot simply return results via the program stack. To work around this issue, we define a continuation interface in lines 4 - 6. The caller must provide an instance of `Continuation` to each I/O request. When the request has completed, the continuation is then called with the result. For convenience, the continuation interface extends the `Callback` defined by the scheduler. In addition to the completion method, `run`, it defines a method for handling exceptions, `error`.

**The `Reader` class**. In lines 7 through 33, we define a `Reader` class, which supports reading one line at a time from a channel. To use the class, one constructs a `Reader` instance and then makes requests through the `readLine` method. If the request is successful, a string representing one line of input is passed to the `run` method of the user-provided continuation.

A `readLine` request is initiated by registering a callback with the scheduler, associating it with the `READ_RDY_EVT` and `ERR_EVT` events. The actual read is performed within this callback.

Unfortunately, reads on a socket-based channel are not guaranteed to return an entire request. Thus, multiple `read` calls may be needed to retrieve a single request. In a multi-threaded or non-concurrent implementation, one would typically implement this using two nested loops, with the outer calling `read` on the channel and the inner loop scanning through the characters that were read, looking for a newline. This code might look as follows:

```
char c = '\0';
while (c!='\n') {
  ch.read(cBuf); // blocking read
  // Copy from the buffer
  while (cBuf.hasRemaining() &&
         (c=cBuf.get())!='\n') {
    sBuf.put(c);
  }
}
```

In an event-driven program, we instead interleave other I/O requests while we wait for data to arrive on the channel. Unfortunately, this means that the outer loop must be broken across multiple callbacks. After each `read` on the channel, the characters read are scanned for a newline (equivalent to the inner loop above). If one is found, the resulting line is passed to the continuation (line 22). Otherwise, the callback is re-registered with the scheduler, awaiting the availability of more data on the channel (line 23). This reregistration creates, in effect, an "outer loop" which continues the register-callback cycle until a line has been read.

Error handling is also more complex than in a non-event implementation. An error on the channel cannot be thrown as an exception but must be instead passed to the continuation. In the `readLine` method, if the event returned was an error, the `error` method of the user's continuation is called (line 29). Then, no further callbacks are registered, ending the processing by this reader class.

**The `write` and `accept` methods**. The implementation of the `EventIO.write` and `EventIO.accept` methods follows the same approach as `readLine`. An initial request is registered with the scheduler. The associated channel operation (`write` or `accept`) may require multiple calls to complete the request. Thus, the callback re-registers itself with the scheduler until the request has completed.

**Using the Event Library** Figure 2 shows a simple web server built on top of this event I/O library. The server contains two loops: one for accepting new connections and a second, per connection, for reading requests and sending responses. Since these loops cross I/O calls, they must be broken into several callbacks. A loop thus becomes a circular sequence of callbacks.

The accept loop is initiated by the `start` method (line 46), which calls `EventIO.accept`, passing a newly created `AcceptCont` continuation. After processing a newly accepted channel, this continuation calls `EventIO.accept` again (line 58), providing itself as a continuation.

The request loop is initiated within `AcceptCont`, which passes a new instance of `ReadCont` to `readLine` (line 57). Upon completion of a read, the read continuation parses the request and reads the associated file (lines 73 - 76). The response is written using `EventIO.write`, which is passed an instance of `WriteCont` as a continuation (line 78). Upon completion of the write, this continuation initiates another read (line 88), passing the original `ReadCont` instance as a continuation, thus starting another iteration of the loop.

The `error` methods of all three continuations log the exception and then close the associated channel. By not initiating another request, they implicitly exit the associated loop.

**Disadvantages of the event-driven implementation** Clearly, it is difficult to follow the control flow of this event driven program. Since a `register` call and the processing of the associated event are broken across two functions (which are called from different places in the program), there is no way to encapsulate

```
45 public class EventWebServer {
46   public static void start(CharChannel acceptCh) {
47     EventIO.accept(acceptCh, new AcceptCont(acceptCh));
48   }
49   static class AcceptCont implements Continuation {
50     CharChannel acceptCh;
51     AcceptCont(CharChannel acceptCh) {
52       this.acceptCh = acceptCh;
53     }
54     public void run(Object retVal) {
55       CharChannel ch = (CharChannel)retVal;
56       EventIO.Reader rdr = new EventIO.Reader(ch);
57       rdr.readLine(new ReadCont(ch, rdr));
58       EventIO.accept(acceptCh, this);
59     }
60     public void error(Exception e) {
61       ... print error message to log ...
62       acceptCh.close();
63     }
64   }
65   static class ReadCont implements Continuation {
66     CharChannel ch; EventIO.Reader rdr;
67     WriteCont writeCont;
68     public ReadCont(CharChannel ch, Reader rdr) {
69       this.ch = ch; this.rdr = rdr;
70       this.writeCont = new WriteCont(this);
71     }
72     public void run(Object retVal) {
73       String filename = parseRequest((String)retVal);
74       try {
75         CharBuffer sendData = readFile(filename);
76       }
77       catch (Exception e) { this.error(e); }
78       EventIO.write(ch, sendData, writeCont);
79     }
80     public void error(Exception e) { ... }
81   }
82   static class WriteCont implements Continuation {
83     ReadCont readCont;
84     public WriteCont(ReadCont readCont) {
85       this.readCont = readCont;
86     }
87     public void run(Object retVal) {
88       readCont.rdr.readLine(readCont);
89     }
90     public void error(Exception e) { ... }
91   }
92 }
```

**Fig. 2.** Event-driven web server

these within a single function call. Also, if we would want to call a given function (e.g., `readLine`) from multiple places, we cannot take advantage of the implicit return behavior of function calls — we must pass the return point to the common function explicitly (e.g., as a callback).

There are two classes of bugs which are much more likely to occur in an event-driven implementation. A *lost continuation* occurs when a callee does not call the continuation passed to it by its caller. For example, if line 22 is left out of the `ReadCb` class in figure 1, the results of a read request will never be processed. If line 23 is left out, the read request will not even be completed. In general, except when a logical flow of control terminates, every control flow path through a callback must end with a call to a continuation or the registration of another callback.

The *lost exception* problem is the corresponding issue for error control flow. Unfortunately, exceptions cannot be used effectively in the above style. Error control flow can be factored into a separate sequence of callbacks (as done using the `error` method in the example), but it is still easy to forget to check for an error or to skip the corresponding continuation call. Potential instances of this problem in the example of figure 1 include not registering for the `ERR_EVT` event (lines 12 and 25) and dropping the error continuation (line 29).

### 2.2 A Web Server in TaskJava

Figure 3 shows a TaskJava I/O library providing the same functionality as the event-driven code in Figure 1. Rather than call continuations to signal completion of a request, `readline`, `write`, and `accept` return values directly to their callers.

**The `wait` function.** Like event-driven systems, TaskJava programs are used with non-blocking I/O libraries and with a user-defined event scheduler. However, rather than forcing programmers to explicitly register a callback to be called when an event occurs, TaskJava provides a primitive `wait` function, which takes a set of event identifiers and suspends the current method until one of the events occurs, at which point the method resumes.

A `wait` call does not cause the program to block at the operating system level. Instead, the containing method is automatically translated into a form of *continuation-passing style* (CPS). The method is broken by the compiler at each `wait` call into two separate methods, one that executes up to the `wait` call and one that continues after the event is signaled. The `wait` call is replaced in the first method by an event registration, as in the event-driven style shown earlier.

**Asynchronous methods.** Methods that contain a call to `wait` must be declared with an `async` modifier. These are called *asynchronous methods*. The modifier indicates to the compiler that it should perform a CPS translation of the method. Asynchronous methods provide programmers with the procedural abstraction that is lacking in explicit event-driven programs. With the exception of the additional annotation, such methods appear to clients as standard Java methods.

```
01 public class TaskIO {
02   public static class Reader {
03     ...
04     public async String readLine() throws IOException {
05       String line;
06       // keep reading until we finish a line
07       do {
08         Event event = wait(ch, Event.READ_RDY_EVT,
09                            Event.ERR_EVT);
10         if (event.type() == Event.READ_RDY_EVT) {
11           ch.read(cbuf);
12           line = scanChars();
13         }
14         else {
15           assert event.type() == Event.ERR_EVT;
16           throw event.getError();
17         }
18       } while (line==null);
19       return line;
20     }
21   public static async void write(CharChannel ch,
22                                  CharBuffer data)
23     throws IOException
24   {
25     while (data.hasRemaining()) {
26       Event event = wait(ch, Event.WRITE_RDY_EVT,
27                          Event.ERR_EVT);
28       if (event.type() == Event.READ_RDY_EVT)
29         ch.write(data);
30       else {
31         assert event.type() == Event.ERR_EVT;
32         throw event.getError();
33       }
34     }
35   }
36   public static CharChannel accept(CharChannel ch)
37     throws IOException { ... }
38   ...
39 }
```

**Fig. 3.** A task-based nonblocking I/O library

If a method calls an asynchronous method, it must also be declared `async`. Since the callee will not return a value directly, the calling method must be translated to use continuation passing style as well. Thus, when a `wait` call completes, results are propagated back up the logical call stack by calling each method's continuation. This behavior is transparent to the programmer. Methods that are not declared `async` do not require any translation. Further, asynchronous methods may call any standard Java method.

**`wait`-based I/O Library Implementation** We now look in more detail at the I/O library of figure 3. The `readLine` method of `TaskIO.Reader` is declared `async` and returns a `String` containing a line of data read from the channel. If an error occurs, this method throws an `IOException`.

Unlike the event-driven case, our interleaved calls to `read` may be contained within a single loop (lines 7 - 18). At the head of this loop, a call to `wait` is made, stopping execution of the method until either the `READ_RDY_EVT` or `ERR_EVT` events occur (line 8). An event object associated with the event is returned by `wait` upon resumption of the method. If a `READ_RDY_EVT` event is returned, the channel is read (line 11) and the resulting buffer scanned for a newline. If a newline is encountered, the loop is exited and the associated line of data returned to the caller. Otherwise, the loop continues until either an entire line has finished or an error occurs.

If an `ERR_EVT` event is returned by `wait`, the associated exception is obtained from the event and then re-thrown to the caller of `readLine` (line 16).

The `TaskIO.write` and `TaskIO.accept` methods are implemented in a similar manner to `readLine`. Both methods are declared `async` and throw an `IOException` to signal an error. The `accept` method returns to its caller a new channel created for the incoming connection. `write` is declared `void` since it does not return a value. It does, however, block the caller until the write has completed.

**Tasks** Figure 4 shows a simple web server built on top of this task-based I/O library, equivalent in functionality to the server of figure 2. As with the event-driven implementation, this server contains two loops: one for accepting new connections and a second, per connection, for reading requests and sending responses. However, these two loops are encapsulated within *tasks*, the units of concurrency for TaskJava programs. Any class that implements the `Task` interface, such as `AcceptTask` and `RequestTask` in figure 4, is considered a task. Much like Java's `Thread` interface, the `Task` interface defines a single method, `run`, which takes no parameters and returns no value. This method may contain calls to `wait` and `async` methods. It should provide the main control flow for the task.

Instances of a task may be created by using the `spawn` keyword, which is followed by a constructor call (the constructor may be either the default constructor or any public constructor declared on the task's class). The spawn of a

```
40 public class TaskWebServer {
41   public static start(CharChannel acceptCh) {
42     spawn AcceptTask(acceptCh);
43   }
44   public static class AcceptTask implements Task {
45     CharChannel acceptCh;
46     AcceptTask(CharChannel acceptCh) {
47       this.acceptCh = acceptCh;
48     }
49     public void run() {
50       try {
51         while (true) {
52           CharChannel ch = TaskIO.accept(acceptCh);
53           spawn RequestTask(ch);
54         }
55       } catch (IOException e) {
56         ... print error message to log ...
57         acceptCh.close();
58       }
59     }
60   }
61   public class RequestTask implements Task {
62   private CharChannel ch;
63   public RequestTask(CharChannel ch) { this.ch = ch; }
64   public void run() {
65     TaskIO.Reader rdr = new TaskIO.Reader(ch);
66     try {
67       while (true) { // main request loop
68         String filename
69            = parseRequest(rdr.readLine());
70         charBuffer sendData = readFile(filename);
71         TaskIO.write(ch, sendData);
72       }
73     } catch (IOException e) {
74       ...
75   }
76 }
```

**Fig. 4.** Implementation of Web Server in TaskJava

task calls the appropriate constructor to create a new object and then the `run` method to start execution of the task.

If a task makes a `wait` call, either directly or through an `async` method, the task is effectively blocked until an event registered by `wait` occurs. However, tasks do not block at the operating system level. Instead the `run` method is translated into continuation passing style. Each asynchronous call is passed a `Callback` object (generated by the TaskJava compiler), which, when called, resumes the task.

**Web server implementation** We will now look at how tasks are used in the web server of figure 4. To start the server, one calls the `start` method, which spawns a new `AcceptTask` task, passing it the channel to be used to listen for new connections (line 42). The body of this task's `run` method contains a loop which calls `TaskIO.accept` to accept a new connection and then spawns a new `RequestTask` task to process requests on the new channel (line 53).

If the `accept` call throws an `IOException`, this is caught at line 55. In this case, an error message is logged and the connection closed. Since the loop has been exited, the `run` method returns, effectively terminating the task.

The `run` method of `RequestTask` first creates a `TaskIO.Reader` for the new channel (line 65). It then enters its main loop for handling requests (lines 68 - 71). The name of the file to retrieve is obtained by calling `rdr.readLine()` to read a line of input and then `parseRequest` to parse the request. The requested file is read and its contents written to the channel using `TaskIO.write`. If an `IOException` is thrown, it is caught at line 74, where the error is logged, the channel closed, and the task terminated. Otherwise, if no error occurs, the next request is read from the channel.

**Benefits of TaskJava** **Simpler control flow**. Unlike the pure event-driven implementation, the TaskJava version has a simple control flow, which is made explicit through standard structured programming constructs (`while`, `for`, etc.). This makes it easy for the programmer as well as automatic tools to reason about the program's behavior.

**Elimination of potential errors**. In addition, the *lost continuation* and *lost exception* problems are not issues in this implementation. Each method call will always return control back to its caller, unless an exception is thrown or the entire program terminated. Standard Java exceptions are used to structure error handling. The TaskJava compiler, like the Java compiler, statically verifies that each thrown exception is explicitly handled.

**Better use of Object Orientation**. The TaskJava web server also has a more natural object-oriented design. The classes used in this implementation (`AcceptTask` and `RequestTask`) have a natural correspondence to the problem domain. In the event-driven version, the request task is broken into two callbacks: `ReadCont` and `WriteCont`. In a more realistic server with more complex error

| | | | |
|---|---|---|---|
| $FJ^+$ | Program | P | ::= $\overline{CL}$ return e; |
| | Class List | CL | ::= class C extends C {$\bar{T}$ $\bar{f}$; K $\bar{M}$} |
| | Constructor | K | ::= C($\bar{T}$ $\bar{f}$) { super($\bar{f}$); this.$\bar{f}$= $\bar{f}$;} |
| | Method | M | ::= T m($\bar{T}$ $\bar{x}$) throws $\bar{C}$ {return e;} |
| | Type | T | ::= C \| Bag<T> |
| | Base Expressions | $e_{base}$ | ::= x \| e.f \| e.m($\bar{e}$) \| new C($\bar{e}$) \| (C)e<br>\| {$\bar{e}$} \| throw e \| try {e;} catch (C x) { e; } |
| | | e | ::= $e_{base}$ |
| EJ | | e | ::= $e_{base}$ \| reg(e, e) |
| CTJ | Async methods | M | ::= ... \| async C m($\bar{C}$ $\bar{x}$) throws $\bar{C}$ {return e;} |
| | | e | ::= $e_{base}$ \| spawn C(e) \| wait(e) |

**Fig. 5.** Syntax of $FJ^+$, EJ, CTJ.

handling, even more callback classes may be needed. For example, the event-driven web server in the case study of section 7 requires six callback classes to process an HTTP request.

As a result, it is much easier to extend request handling in the TaskJava web server. For example, to support different protocols and content types, one might use the *template method* design pattern [10]. Key steps in the processing of a request (e.g., reading the request, obtaining the response, sending the response) might be factored out into separate methods on `RequestTask`. The `run` method now calls these template methods at the appropriate points in request processing. To change implementation of some step in a request, one subclasses from `RequestTask` and overrides the appropriate method. For example, to support SSL, one would override the methods which read the request and write the response. The new read method would call the superclass's read method and then decrypt the request. The new write method would encrypt the response and then call the superclass's write method.

It is unclear how a similar extensibility scheme could be implemented in an event-driven server, given the need to break control flow over many continuation classes.

## 3 Formal Model

We formalize TaskJava and prove our theorems in a core calculus extending Featherweight Java (FJ) [14]. We do this in three steps: first, we define $FJ^+$, an extension to FJ with exceptions and built-in multiset data structures; second, we define EventJava (EJ), a core calculus for event-driven programs into which tasks will be compiled; and finally, we define the core features of TaskJava in CoreTaskJava (CTJ).

### 3.1 FJ$^+$

The syntactic elements of FJ$^+$ are described in Figure 5. An FJ$^+$ program consists of a *class table* mapping class names to classes, and an initial expression. As in FJ, the notation $\bar{\texttt{D}}$ denotes a sequence of elements from domain `D`. A class consists of a list of typed *fields*, a *constructor*, and a list of typed *methods*. The metavariable `C` ranges over class names, `f` over field names, `m` over method names, and `x` over formal parameter names. An expression is either a formal, a field access, a method call, an object allocation, a type cast, a set, the `throw` of an exception, or a `try` expression. We assume there exist built-in classes `Object` and `Throwable`. The class `Throwable` is a subclass of `Object` and both have no fields and no methods.

We define the operational semantics of FJ$^+$ as additions to the rules of FJ [1] and then EventJava and CoreTaskJava as mutually exclusive additions to these rules.

Program execution is modeled as a sequence of rewritings of the initial expression, which either terminates when a value is obtained or diverges if the rewritings never yield a value. For all three languages, programs evaluate to either non-exception values of the form $v ::= \texttt{new}\ C() \mid \texttt{new}\ C(\bar{\texttt{v}}) \mid \{\bar{\texttt{v}}\}$ or exception values of the form $\texttt{throw new}\ C_e(\bar{\texttt{v}})$, where $C_e <: \texttt{Throwable}$. In the evaluation and typing rules, we use $v$ as shorthand for a non-exception value, $\bar{\texttt{v}}$ for a sequence of non-exception values, and $v_e$ for the non-exception value $\texttt{new}\ C(\bar{\texttt{v}})$.

Figures 7 and 8 list the operational rules of FJ$^+$. We use the symbol $E$ to represent an **evaluation context**, i.e., an expression where the next subexpression to be evaluated (using a leftmost, call-by-value ordering) has been replaced with a placeholder []. Formally,

$$\begin{aligned} E ::= {} & [] \mid E.f \mid E.m(e) \mid v.m(\bar{\texttt{v}}, E, \bar{\texttt{e}}) \mid \texttt{new}\ C(\bar{\texttt{v}}, E, \bar{\texttt{e}}) \\ & \mid \{\bar{\texttt{v}}, E, \bar{\texttt{e}}\} \mid (C)E \mid \texttt{throw}\ E \mid \texttt{try}\ \{E;\}\ \overline{\texttt{CK}} \end{aligned}$$

We write $E[e]$ to represent the expression created by substituting the subexpression $e$ for the placeholder in the evaluation context $E$. Evaluation contexts are used in the evaluation rules, the type soundness theorems, and the translation relation.

As in Featherweight Java, the computation rules for cast only permit progress when the type of the value being cast is a subtype of the target type. Otherwise, the computation becomes "stuck". The subtyping relation, defined in Figure 6, is extended to include bags (rule S-Bag) and exception effects (rule S-Exc). Based on the subtyping relation, we define a join operator $\sqcup$ on types, where $\texttt{T} \sqcup \texttt{T}'$ is the least upper bound of types `T` and `T'`. It is extended to sets of types in the

[1] We use the rules in chapter 19 of *Types and Programming Languages* [20] rather than those of the original FJ paper [14], as they provide deterministic, call-by-value evaluation.

$\boxed{\mathtt{T} <: \mathtt{T}'}$

$$\frac{}{\mathtt{T} <: \mathtt{T}} \text{ (S-Self)} \qquad \frac{\mathtt{T} <: \mathtt{T}' \qquad \mathtt{T}' <: \mathtt{T}''}{\mathtt{T} <: \mathtt{T}''} \text{ (S-Trans)}$$

$$\frac{CT(C) = \mathtt{class}\ C\ \mathtt{extends}\ D\ \{...\}}{C <: D} \text{ (S-Cls)} \qquad \frac{\mathtt{T} <: \mathtt{T}'}{\mathtt{Bag} < \mathtt{T} > \ <: \ \mathtt{Bag} < \mathtt{T}' >} \text{ (S-Bag)}$$

$\boxed{\mathtt{T}|\bar{\tau} <: \mathtt{T}'|\bar{\tau}'}$

$$\frac{\mathtt{T} <: \mathtt{T}' \qquad \bar{\tau} \subseteq: \bar{\tau}'}{\mathtt{T}|\bar{\tau} <: \mathtt{T}'|\bar{\tau}'} \text{ (S-Exc)}$$

**Fig. 6.** Subtyping rules for FJ$^+$, EJ, and CTJ.

obvious way. We write $\sqcup\bar{\mathtt{T}}$ to denote the least upper bound of all types in the set $\bar{\mathtt{T}}$. The join operator is undefined for joins between class types and bag types.

$\boxed{e \longrightarrow e'}$

$$\frac{fields(C) = \bar{\mathtt{T}}\ \bar{\mathtt{f}}}{(\mathtt{new}\ C(\bar{\mathtt{v}})).f_i \longrightarrow v_i} \text{ (E-1)} \qquad \frac{mbody(m, C) = (\bar{\mathtt{x}}, e_0)}{(\mathtt{new}\ C(\bar{\mathtt{v}})).m(\bar{\mathtt{v}}_e) \longrightarrow [\bar{\mathtt{v}}_e/\bar{\mathtt{x}}, \mathtt{new}\ C(\bar{\mathtt{v}})/\mathtt{this}]e_0} \text{ (E-2)}$$

$$\frac{C <: C'}{(C')(\mathtt{new}\ C(\bar{\mathtt{v}})) \longrightarrow \mathtt{new}\ C(\bar{\mathtt{v}})} \text{ (E-3)} \qquad \frac{\forall \mathtt{T}_i \in \bar{\mathtt{T}}.\mathtt{T}_i <: \mathtt{T}}{(\mathtt{Bag} < \mathtt{T} >)\{\bar{\mathtt{v}}\} \longrightarrow \{\bar{\mathtt{v}}\}} \text{ (E-4)}$$

$\mathtt{new}\ C(\bar{\mathtt{v}}, \mathtt{throw}\ v_e, \bar{\mathtt{e}}) \longrightarrow \mathtt{throw}\ v_e$ (E-5) $\quad (\mathtt{throw}\ v_e).m(\bar{\mathtt{e}}) \longrightarrow \mathtt{throw}\ v_e$ (E-6)

$v.m(\bar{\mathtt{v}}, \mathtt{throw}\ v_e, \bar{\mathtt{e}}) \longrightarrow \mathtt{throw}\ v_e$ (E-7) $\quad \{\bar{\mathtt{v}}, \mathtt{throw}\ v_e, \bar{\mathtt{e}}\} \longrightarrow \mathtt{throw}\ v_e$ (E-8)

$(\mathtt{throw}\ v_e).f \longrightarrow \mathtt{throw}\ v_e$ (E-9) $\quad (C)(\mathtt{throw}\ v_e) \longrightarrow \mathtt{throw}\ v_e$ (E-10)

$\mathtt{throw}\ \mathtt{throw}\ v_e \longrightarrow \mathtt{throw}\ v_e$ (E-11) $\quad \mathtt{try}\ \{v;\}\ \overline{\mathtt{CK}} \longrightarrow v$ (E-12)

$$\frac{v = \mathtt{new}\ C(\bar{\mathtt{v}}) \qquad C <: C_e}{\mathtt{try}\ \{\mathtt{throw}\ v;\}\ \mathtt{catch}\ (C_e\ x)\ \{e;\} \longrightarrow [v/x]e} \text{ (E-13)} \qquad \frac{v = \mathtt{new}\ C(\bar{\mathtt{v}}) \qquad C \not<: C_e}{\mathtt{try}\ \{\mathtt{throw}\ v;\}\ \mathtt{catch}\ (C_e\ x)\ \{e;\} \longrightarrow \mathtt{throw}\ v} \text{ (E-14)}$$

**Fig. 7.** Computation rules for FJ$^+$.

### 3.2 EventJava

EventJava (EJ) is a core calculus that extends FJ$^+$ with support for events and event registration. Figure 5 gives the syntax for EJ, showing the extensions from FJ$^+$. The set of EJ expressions additionally contains a built-in function `reg`,

$\boxed{e \longrightarrow e'}$

$$\frac{e_0 \longrightarrow e_0'}{e_0.f \longrightarrow e_0'.f}\ \text{(E-15)} \qquad \frac{e_0 \longrightarrow e_0'}{e_0.m(\bar{\mathtt{e}}) \longrightarrow e_0'.m(\bar{\mathtt{e}})}\ \text{(E-16)} \qquad \frac{e_i \longrightarrow e_i'}{v_0.m(\bar{\mathtt{v}}, e_i, \bar{\mathtt{e}}) \longrightarrow v_0.m(\bar{\mathtt{v}}, e_i', \bar{\mathtt{e}})}\ \text{(E-17)}$$

$$\frac{e_i \longrightarrow e_i'}{\mathtt{new}\ C(\bar{\mathtt{v}}, e_i, \bar{\mathtt{e}}) \longrightarrow \mathtt{new}\ C(\bar{\mathtt{v}}, e_i', \bar{\mathtt{e}})}\ \text{(E-18)} \qquad \frac{e_0 \longrightarrow e_0'}{(T)e_0 \longrightarrow (T)e_0'}\ \text{(E-19)} \qquad \frac{e_0 \longrightarrow e_0'}{\{\bar{\mathtt{v}}, e_0, \bar{\mathtt{e}}\} \longrightarrow \{\bar{\mathtt{v}}, e_0', \bar{\mathtt{e}}\}}\ \text{(E-20)}$$

$$\frac{e_0 \longrightarrow e_0'}{\mathtt{throw}\ e_0 \longrightarrow \mathtt{throw}\ e_0'}\ \text{(E-21)} \qquad \frac{e_t \longrightarrow e_t'}{\mathtt{try}\ \{e_t;\}\ \mathtt{catch}\ (C_e\ x)\ \{e;\} \longrightarrow \mathtt{try}\ \{e_t';\}\mathtt{catch}\ (C_e\ x)\ \{e;\}}\ \text{(E-22)}$$

**Fig. 8.** Congruence rules for FJ$^+$.

which registers a set of events and a callback with the scheduler. For use with the `reg` function, we assume the system scheduler implementation includes a class `Event`. Further, EJ provides a built-in class `Callback`:

```
class Callback extends Object {
  Object run(Object retVal) { return new Object(); } }
```

The type signature of `reg` is `Bag` $<$ `Event` $>$ $\times$ `Callback` $\rightarrow$ `Object`.

The operational semantics of EJ programs is given with respect to a *program state*. An EventJava program state $\sigma_e$ consists of (1) an expression representing the in-progress evaluation of the currently executing callback, and (2) a bag $\mathcal{E}$ of pending event registrations of type `Bag` $<$ `Event` $>$ $\times$ `Callback`. In the operational rules, we write this state as $e|\mathcal{E}$.

After the initialization expression has been evaluated, the program enters an *event processing loop*. The event processing loop runs until the set of event registrations $\mathcal{E}$ is empty. In each iteration of the loop, one event registration $(s, c)$ is nondeterministically removed from $\mathcal{E}$. An event $\eta$ is then nondeterministically chosen from $s$, and the callback function $c.\mathtt{run}(\eta)$ is executed. A registration is one-time; after the selection of an event $\eta$ , the entire event set $s$ is excluded from further consideration, unless the set is explicitly re-registered with the scheduler. If an empty set of events is passed along with a callback, the callback is guaranteed to be called at some point in the future. An instance of `NullEvent` is then passed to the callback. This semantics models a single-threaded event server.

Note that the parameter of a callback's `run` method has a type of `Object`, even though it will be passed an `Event`. This slight discrepancy simplifies the translation between CTJ (which also uses callbacks for completion of asynchronous method calls) and EJ. As a result, the body of each callback class must downcast the `retVal` parameter to `Event`. Downcasts could be avoided by extending EJ and CTJ with generics, as in Featherweight Generic Java [14].

We define evaluation contexts for EventJava in the same manner as FJ$^+$. The grammar for evaluation contexts has the following extensions for syntactic

$\boxed{e \longrightarrow_e e'}$

Figure 7, rules E-1 - E-14.
Figure 8, rules E-15 - E-22.

$$\frac{e_0 \longrightarrow_e e_0'}{\texttt{reg}\ e_0, e_1 \longrightarrow_e \texttt{reg}\ e_0', e_1}\ (E_e\text{-}23) \qquad \frac{e_0 \longrightarrow_e e_0'}{\texttt{reg}\ v, e_0 \longrightarrow_e \texttt{reg}\ v, e_0'}\ (E_e\text{-}24)$$

$$\texttt{reg throw}\ v, e \longrightarrow_e \texttt{throw}\ v\ (E_e\text{-}25) \qquad \texttt{reg}\ v_0, \texttt{throw}\ v_1 \longrightarrow_e \texttt{throw}\ v_1\ (E_e\text{-}26)$$

$\boxed{e|\mathcal{E} \overset{l}{\Longrightarrow}_e e'|\mathcal{E}'}$

$$\frac{e \longrightarrow_e e'}{e|\mathcal{E} \overset{\epsilon}{\Longrightarrow}_e e'|\mathcal{E}}\ (E_e\text{-Con}) \qquad \frac{}{E[\texttt{reg}\ v_0, v_1]|\mathcal{E} \overset{v_0}{\Longrightarrow}_e E[v_1]|\mathcal{E} \cup (v_0, v_1)}\ (E_e\text{-Reg})$$

$$\frac{(s, v_{cb}) \in \mathcal{E} \qquad \eta \in s}{v|\mathcal{E} \overset{\eta}{\Longrightarrow}_e v_{cb}.\texttt{run}(\eta)|\mathcal{E} \setminus (s, v_{cb})}\ (E_e\text{-Run}) \qquad \frac{(\emptyset, v_{cb}) \in \mathcal{E} \qquad \eta_0 = \texttt{new NullEvent}()}{v|\mathcal{E} \overset{\eta_0}{\Longrightarrow}_e v_{cb}.\texttt{run}(\eta_0)|\mathcal{E} \setminus (\emptyset, v_{cb})}\ (E_e\text{-}\eta_\emptyset\text{Run})$$

$$\texttt{throw}\ v_e|\mathcal{E} \overset{\epsilon}{\Longrightarrow}_e \texttt{throw}\ v_e|\emptyset\ (E_e\text{-Throw})$$

**Fig. 9.** Operational Semantics of EJ.

forms specific to EventJava:

$$E ::= \ \ldots \mid \texttt{reg}(E, e) \mid \texttt{reg}(\{\bar{\texttt{v}}\}, E)$$

Figure 9 lists the operational rules unique to EventJava. In these rules, $s$ ranges over event sets and $\eta$ ranges over events. We define two evaluation relations for EventJava programs. The $\longrightarrow_e$ relation extends the FJ$^+$ $\longrightarrow$ relation with congruence rules to evaluate the arguments of a `reg` call. The $\Longrightarrow_e$ relation then extends this relation to EventJava program states. Rule $E_e$-Con incorporates the $\longrightarrow$ relation into the context of a program state.

Each transition rule of the $\Longrightarrow_e$ relation has an associated **observable action**, denoted by a label above the transition arrow. This action represents the impact of the transition on the scheduler. A label may be either: (1) an event set, representing a registration with the scheduler, (2) a single event, representing the selection of an event by the scheduler, or (3) $\epsilon$, representing a transition which has no impact on the scheduler's state.

### 3.3 CoreTaskJava

The syntax of CoreTaskJava (CTJ) extends FJ$^+$ with three new constructs: the `spawn` syntax for creating tasks; a `wait` primitive, which accepts a set of events

and blocks until one of them occurs; and asynchronous methods via the `async` modifier. A task in CTJ subclasses from a built-in class `Task`:

```
class Task extends Callback {}
```

A task's `run` method contains the body of the task and is called after a task is spawned. [2]

Informally, tasks are modeled as concurrent threads of execution that block when waiting for an asynchronous call to complete. We define evaluation contexts for CTJ expressions in the same manner as FJ$^+$. The grammar for evaluation contexts has the following extensions for syntactic forms specific to CTJ:

$$E ::= ... \mid \texttt{wait}\ E \mid \texttt{spawn}\ E$$

As with EJ, the semantics is defined with respect to a *program state*. A *CTJ program state* $\sigma_c$ consists of: (1) an expression representing the in-progress evaluation of the currently executing task, and (2) a set $\mathcal{B}$ of $(\texttt{Bag} < \texttt{Event} >, E[])$ pairs representing the evaluation contexts of blocked tasks and the events that each task is blocked on. In the operational rules, we write this state as $e|\mathcal{B}$.

After a CTJ program's initialization expression has been evaluated, the program nondeterministically selects a task from the blocked set, then nondeterministically selects an event from the task's event set, and evaluates the task until it either terminates or blocks again. Another task is then selected nondeterministically. This process repeats until the program reaches the state where the current expression is a value and the blocked set is empty. Tasks and the associated event sets are added to $\mathcal{B}$ through calls to `wait`. In addition, the `spawn` of a task is modeled by placing the task in $\mathcal{B}$ with an empty event set, ensuring that the task will eventually be scheduled.

Figure 10 lists the operational rules for CoreTaskJava. As with EventJava, the rules are written using two relations. The $\longrightarrow_c$ relation extends the FJ$^+$ $\longrightarrow$ relation with congruence rules to evaluate the arguments of `wait` and `spawn` calls. The $\Longrightarrow_c$ relation (defined in figure 10) then extends this relation to CoreTaskJava program states. The evaluation of the $\longrightarrow$ rules in the context of a program state is handled by Rule $E_c$-Con. Each transition of the $\Longrightarrow$ relation is labeled with an *observable action*, as defined above for EventJava.

### 3.4 Type Checking

**FJ$^+$ and EJ** Typing judgments for expressions in FJ$^+$ have the form $\Gamma \vdash e : T|\bar{\tau}$, where the environment $\Gamma$ maps variables to types, $T$ is a type (using the type grammar in Figure 5), and $\bar{\tau}$ is set of exception classes (`Throwable` or a

[2] In the implementation, `Task` does not subclass from `Callback` and its `run` method takes no parameters. We subclass from `Callback` here to simplify the presentation of the formalization.

$\boxed{e \longrightarrow_c e'}$

Figure 7, rules E-1 - E-14.
Figure 8, rules E-15 - E-22.

$$\frac{e_0 \longrightarrow_c e_0'}{\texttt{wait}\ e_0 \longrightarrow_c \texttt{wait}\ e_0'}\ (E_c\text{-}23) \qquad \texttt{wait throw}\ v \longrightarrow_c \texttt{throw}\ v\ (E_c\text{-}24)$$

$$\frac{e_0 \longrightarrow_c e_0'}{\texttt{spawn}\ e_0 \longrightarrow_c \texttt{spawn}\ e_0'}\ (E_c\text{-}25) \qquad \texttt{spawn throw}\ v \longrightarrow_c \texttt{throw}\ v\ (E_c\text{-}26)$$

$\boxed{e|\mathcal{B} \stackrel{l}{\Longrightarrow}_c e'|\mathcal{B}'}$

$$\frac{e \longrightarrow_c e'}{e|\mathcal{B} \stackrel{\epsilon}{\Longrightarrow}_c e'|\mathcal{B}}\ (E_c\text{-Con}) \qquad \texttt{throw}\ v|\mathcal{B} \stackrel{\epsilon}{\Longrightarrow}_c \texttt{throw}\ v|\emptyset\ (E_c\text{-Throw})$$

$$\frac{w = \texttt{wait}\ \{\bar{\texttt{v}}\}}{E[w]|\mathcal{B} \stackrel{\{\bar{\texttt{v}}\}}{\Longrightarrow}_c \texttt{new Object}()|\mathcal{B} \cup (\{\bar{\texttt{v}}\}, E[])}\ (E_c\text{-Wait}) \qquad \frac{(\{\bar{\texttt{v}}\}, E[]) \in \mathcal{B} \qquad \eta \in \{\bar{\texttt{v}}\}}{v|\mathcal{B} \stackrel{\eta}{\Longrightarrow}_c E[\eta]|\mathcal{B} \setminus (\{\bar{\texttt{v}}\}, E[])}\ (E_c\text{-Run})$$

$$\frac{(\emptyset, E[]) \in \mathcal{B} \qquad \eta_0 = \texttt{new NullEvent}()}{v|\mathcal{B} \stackrel{\eta_0}{\Longrightarrow}_c E[\eta_0]|\mathcal{B} \setminus (\emptyset, E[])}\ (E_c\text{-}\eta_\emptyset\text{Run}) \qquad \frac{E[\texttt{spawn}\ C(\bar{\texttt{v}})]|\mathcal{B} \stackrel{\emptyset}{\Longrightarrow}_c E[\texttt{new}\ C(\bar{\texttt{v}})]|\mathcal{B} \cup (\emptyset, (\texttt{new}\ C(\bar{\texttt{v}})).\texttt{run}([]))}\ (E_c\text{-Spn})$$

**Fig. 10.** Operational Semantics of CTJ.

subclass of `Throwable`). For checking exceptions, we introduce a new relation $\bar{\tau} \subseteq: \bar{\tau}'$, which is true when,for each class $C \in \bar{\tau}$, there exists a class $C' \in \bar{\tau}'$ such that $C <: C'$.

Figure 11 lists the typing rules for FJ$^+$. These rules extend the typing rules of FJ by adding typing of bags and tracking the set of exceptions $\bar{\tau}$ which may be thrown by an expression. In particular, rule T-9 assigns an arbitrary type `T` to a throw statement, based on the statement's context.

The *override* function (rule T-11) is changed to check that the set of exceptions thrown by a method's body is a subset of those declared to be thrown by the method signature. The auxiliary function $mtype(C, m)$ returns the type signature of method $m$ in class $C$. A method's type signature has the form: $\bar{\texttt{T}}_f \rightarrow \texttt{T}_r\ \texttt{throws}\ \bar{\tau}$, where $\bar{\texttt{T}}_f$ represents the types of the formal arguments, $\texttt{T}_r$ represents the type of the return value, and $\bar{\tau}$ represents the exceptions declared to be thrown by the method. EventJava extends the rules of FJ$^+$ with an additional rule to assign a type to `reg` expressions (figure 12).

**CoreTaskJava** Typing rules for CTJ are listed in figures 13 and 14. Typing statements for expressions in CTJ have the form $\Gamma, C, M \vdash e : T|\bar{\tau}$, where $C$ is

$\boxed{\Gamma \vdash e : T|\bar{\tau}}$

$$\frac{x : \mathtt{T} \in \Gamma}{\Gamma \vdash x : \mathtt{T}|\emptyset}\ \text{(T-1)} \qquad \frac{\Gamma \vdash e_0 : C_0|\bar{\tau} \qquad \mathit{fields}(C_0) = \bar{\mathtt{T}}\ \bar{\mathtt{f}}}{\Gamma \vdash e_0.f_i : \mathtt{T}_i|\bar{\tau}}\ \text{(T-2)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_0 : C_0|\bar{\tau}_0 \\ \mathit{mtype}(m, C_0) = \bar{\mathtt{T}}_f \to \mathtt{T}_r\ \mathtt{throws}\ \bar{\tau}_m \qquad \Gamma \vdash \bar{\mathtt{e}} : \bar{\mathtt{T}}_a|\bar{\tau}_a \qquad \bar{\mathtt{T}}_a <: \bar{\mathtt{T}}_f\end{array}}{\Gamma \vdash e_0.m(\bar{\mathtt{e}}) : \mathtt{T}_r|\bar{\tau}_0 \cup \bar{\tau}_m \cup \bar{\tau}_a}\ \text{(T-3)}$$

$$\frac{\mathit{fields}(C_c) = \bar{\mathtt{T}}_c\ \bar{\mathtt{f}} \qquad \Gamma \vdash \bar{\mathtt{e}} : \bar{\mathtt{T}}_e|\bar{\tau} \qquad \bar{\mathtt{T}}_e <: \bar{\mathtt{T}}_c}{\Gamma \vdash \mathtt{new}\ C_c(\bar{\mathtt{e}}) : C_c|\bar{\tau}}\ \text{(T-4)} \qquad \frac{\Gamma \vdash e_0 : \mathtt{T}_0|\bar{\tau} \qquad \mathtt{T}_0 <: \mathtt{T}}{\Gamma \vdash (\mathtt{T})e_0 : \mathtt{T}|\bar{\tau}}\ \text{(T-5)}$$

$$\frac{\Gamma \vdash e_0 : \mathtt{T}_0|\bar{\tau} \qquad \mathtt{T} <: \mathtt{T}_0 \qquad \mathtt{T} \neq \mathtt{T}_0}{\Gamma \vdash (\mathtt{T})e_0 : \mathtt{T}|\bar{\tau}}\ \text{(T-6)} \qquad \frac{\Gamma \vdash e_0 : \mathtt{T}_0|\bar{\tau} \qquad \mathtt{T} \not<: \mathtt{T}_0 \qquad \mathtt{T}_0 \not<: \mathtt{T} \qquad \textit{stupid warning}}{\Gamma \vdash (\mathtt{T})e_0 : \mathtt{T}|\bar{\tau}}\ \text{(T-7)}$$

$$\frac{\Gamma \vdash \forall e_i \in \bar{\mathtt{e}}\ .\ e_i : \mathtt{T}_i \qquad \mathtt{T} = \sqcup \mathtt{T}_i \qquad \bar{\tau} = \cup \bar{\tau}_i}{\Gamma \vdash \{\bar{\mathtt{e}}\} : \mathtt{Bag} < \mathtt{T} > |\bar{\tau}}\ \text{(T-8)} \qquad \frac{\Gamma \vdash e_0 : C_0|\bar{\tau} \qquad C_0 <: \mathtt{Throwable}}{\Gamma \vdash \mathtt{throw}\ e_0 : \mathtt{T}|\bar{\tau} \cup C_0}\ \text{(T-9)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_t : \mathtt{T}_t|\bar{\tau}_t \\ C_e <: \mathtt{Exception} \qquad \Gamma \cup x : C_e \vdash e_c : \mathtt{T}_c|\bar{\tau}_c \qquad \bar{\tau}_t' = \{\tau \in \bar{\tau}_t \mid \tau \not<: \mathtt{C}_e\}\end{array}}{\Gamma \vdash \mathtt{try}\ \{e_t;\}\ \mathtt{catch}\ (C_e\ x)\ \{e_c;\}\ :\ \mathtt{T}_t \sqcup \mathtt{T}_c|\bar{\tau}_t' \cup \bar{\tau}_c}\ \text{(T-10)}$$

$\boxed{\mathit{override}(m, D, \bar{\mathtt{T}}_c \to \mathtt{T}_{rc}, \bar{\tau})}$

$$\frac{\begin{array}{c}\mathit{mtype}(m, D) = \bar{\mathtt{T}}_d \to \mathtt{T}_{rd}\ \mathtt{throws}\ \bar{\tau}_d \Rightarrow \\ \bar{\mathtt{T}}_c = \bar{\mathtt{T}}_d\ \wedge\ \mathtt{T}_{rc} = \mathtt{T}_{rd}\ \wedge\ \bar{\tau} \subset: \bar{\tau}_d\end{array}}{\mathit{override}(m, D, \bar{\mathtt{T}}_c \to \mathtt{T}_{rc}, \bar{\tau})}\ \text{(T-11)}$$

$\boxed{\text{M OK in C}}$

$$\frac{\begin{array}{ccc}\bar{\tau}_e \subset: \bar{\tau} & \{\bar{\mathtt{x}} : \bar{\mathtt{T}}|\emptyset, \mathtt{this} : C|\emptyset\} \vdash e_0 : \mathtt{T}_e|\bar{\tau}_e & \mathtt{T}_e <: \mathtt{T}_r \\ & CT(C) = \mathtt{class}\ C\ \mathtt{extends}\ D\ \{...\} & \mathit{override}(m, D, \bar{\mathtt{T}} \to \mathtt{T}_r, \bar{\tau})\end{array}}{\mathtt{T}_r\ m(\bar{\mathtt{T}}\ \bar{\mathtt{x}})\ \mathtt{throws}\ \bar{\tau}\ \{\mathtt{return}\ e_0;\}\ \text{OK in C}}\ \text{(T-12)}$$

$\boxed{\text{C OK}}$ $\qquad$ $\boxed{\text{P OK}}$

$$\frac{\begin{array}{c}K = C(\bar{\mathtt{T}}_s\ \bar{\mathtt{f}}_s, \bar{\mathtt{T}}_c\ \bar{\mathtt{f}}_c)\{\mathtt{super}(\bar{\mathtt{f}}_s); \mathtt{this}.\bar{\mathtt{f}}_c = \bar{\mathtt{f}}_c;\} \\ \mathit{fields}(C_s) = \bar{\mathtt{T}}_s\ \bar{\mathtt{f}}_s \\ \overline{M}\ \text{OK in C}\end{array}}{\mathtt{class}\ C\ \mathtt{extends}\ C_s\ \{\bar{\mathtt{T}}_c\ \bar{\mathtt{f}}_c;\ K\ \overline{M}\}\ \text{OK}}\ \text{(T-13)} \qquad \frac{\vdash e : \mathtt{T}|\emptyset \qquad \overline{CL}\ \text{OK}}{\vdash^f \overline{CL}\ \mathtt{return}\ e\ \text{OK}}\ \text{(T-14)}$$

**Fig. 11.** Typing rules for FJ$^+$.

$\boxed{\Gamma \vdash e : T|\bar{\tau}}$

Figure 11, rules T-1 - T-13.

$$\frac{\Gamma \vdash e_s : \texttt{Bag} < \texttt{Event} > |\bar{\tau}_s \qquad \Gamma \vdash e_c : \texttt{Callback}|\bar{\tau}_c}{\Gamma \vdash \texttt{reg}\ e_s, e_c : \texttt{Event}|\bar{\tau}_s \cup \bar{\tau}_c}\ (T_e\text{-}14)$$

$\boxed{\text{P OK}}$ $\qquad$ $\boxed{\mathcal{E}\ \text{OK}}$

$$\frac{\vdash e : \texttt{T}|\emptyset \qquad \overline{CL}\ \text{OK}}{\vdash^e \overline{CL}\ \texttt{return}\ e\ \text{OK}}\ (T_e\text{-}15) \qquad \frac{\begin{array}{c}\vdash \{\bar{\texttt{v}}_e\} : \texttt{Bag} < \texttt{Event} > \\ \vdash \texttt{new}\ C(\bar{\texttt{v}}_c) : C|\emptyset \\ C <: \texttt{Callback}\end{array}}{\vdash \{(\{\bar{\texttt{v}}_e\}, \texttt{new}\ C(\bar{\texttt{v}}_c))\}\ \text{OK}}\ (T_e\text{-}16)$$

**Fig. 12.** Typing rules for EJ.

the name of the enclosing class and $M$ the definition of the enclosing method. The auxiliary function $isaync(M)$ returns `true` if the method definition $M$ has an `async` modifier and `false` otherwise. Likewise, $isasync(C, m)$ returns true if the definition of method $m$ in class $C$ has an `async` modifier.

Rules $T_c$-3 and $T_c$-15 ensure that asynchronous calls and `wait` calls may only be made by asynchronous methods or the `run` method of a task. Rules T-11 and T-12 of $FJ^+$ have each been split into two cases, one for asynchronous methods and one for standard methods. This ensures that asynchronous methods may only override asynchronous methods and non-asynchronous methods may only override non-asynchronous methods. Rule $T_c$-12b also verifies that the `run` method of a task has not been declared `async`.

Rule $T_c$-14 ensures that sets of blocked tasks are well-formed. If $\vdash \mathcal{B}$ OK, then $\mathcal{B}$ consists of a set of pairs, where the first element of the pair is a set of events and the second element of the pair is an evaluation context $E[]$. The evaluation context must be well-typed when [] is replaced with an event.

## 4 Properties of CoreTaskJava programs

We now describe the central formal properties of CTJ programs: subject reduction, progress, and soundness.

We start by defining *normal forms* for CTJ and EJ, special forms for expressions obtained by rewriting until no rewrite rule from $\longrightarrow_e$ or $\longrightarrow_c$ is possible. A CTJ expression $e$ is in *normal form* if it matches one of the following forms: $E[\texttt{spawn}\ C(\bar{\texttt{v}})]$, $E[\texttt{wait}\ v]$, $E[(T)v]$ where the type of non-exception value $v$ is not a subtype of $T$, `throw` $v_e$, or a non-exception value $v$. Similarly, an EJ expression $e$ is in normal form if it matches one of the following forms: $E[\texttt{reg}\ v_1, v_2]$, $E[(T)v]$ where the type of non-exception value $v$ is not a subtype of $T$, `throw` $v_e$, or a non-exception value $v$.

$\boxed{\Gamma, C, M \vdash e : T|\bar{\tau}}$

$$\frac{x : T \in \Gamma}{\Gamma, C, M \vdash x : T|\emptyset} \ (\mathrm{T}_c\text{-}1)$$

$$\frac{\Gamma, C, M \vdash e_0 : C_0|\bar{\tau} \qquad fields(C_0) = \bar{\mathtt{T}}\ \bar{\mathtt{f}}}{\Gamma, C, M \vdash e_0.f_i : \mathtt{T}_i|\bar{\tau}} \ (\mathrm{T}_c\text{-}2)$$

$$\frac{\begin{array}{c}\Gamma, C, M \vdash e_0 : C_0|\bar{\tau}_0 \\ mtype(m, C_0) = \bar{\mathtt{T}}_f \to \mathtt{T}_r\ \mathtt{throws}\ \bar{\tau}_m \qquad \Gamma, C, M \vdash \bar{\mathtt{e}} : \bar{\mathtt{T}}_a|\bar{\tau}_a \qquad \bar{\mathtt{T}}_a <: \bar{\mathtt{T}}_f \\ isasync(m, C_0) \implies \\ (C <: \mathtt{Task} \wedge methname(M) = \mathtt{run}) \ \vee \ isasync(M)\end{array}}{\Gamma, C, M \vdash e_0.m(\bar{\mathtt{e}}) : \mathtt{T}_r|\bar{\tau}_0 \cup \bar{\tau}_m \cup \bar{\tau}_a} \ (\mathrm{T}_c\text{-}3)$$

$$\frac{fields(C_c) = \bar{\mathtt{T}}_c\ \bar{\mathtt{f}} \qquad \Gamma, C, M \vdash \bar{\mathtt{e}} : \bar{\mathtt{T}}_e|\bar{\tau} \qquad \bar{\mathtt{T}}_e <: \bar{\mathtt{T}}_c}{\Gamma, C, M \vdash \mathtt{new}\ C_c(\bar{\mathtt{e}}) : C_c|\bar{\tau}} \ (\mathrm{T}_c\text{-}4)$$

$$\frac{\Gamma, C, M \vdash e_0 : \mathtt{T}_0|\bar{\tau} \qquad \mathtt{T}_0 <: \mathtt{T}}{\Gamma, C, M \vdash (\mathtt{T})e_0 : \mathtt{T}|\bar{\tau}} \ (\mathrm{T}_c\text{-}5)$$

$$\frac{\Gamma, C, M \vdash e_0 : \mathtt{T}_0|\bar{\tau} \qquad \mathtt{T} <: \mathtt{T}_0 \qquad \mathtt{T} \neq \mathtt{T}_0}{\Gamma, C, M \vdash (\mathtt{T})e_0 : \mathtt{T}|\bar{\tau}} \ (\mathrm{T}_c\text{-}6)$$

$$\frac{\Gamma, C, M \vdash e_0 : \mathtt{T}_0|\bar{\tau} \qquad \mathtt{T} \not<: \mathtt{T}_0 \qquad \mathtt{T}_0 \not<: \mathtt{T} \qquad stupid\ warning}{\Gamma, C, M \vdash (\mathtt{T})e_0 : \mathtt{T}|\bar{\tau}} \ (\mathrm{T}_c\text{-}7)$$

$$\frac{\Gamma, C, M \vdash \forall e_i \in \bar{\mathtt{e}}\ .\ e_i : \mathtt{T}_i \qquad \mathtt{T} = \sqcup \mathtt{T}_i \qquad \bar{\tau} = \cup \bar{\tau}_i}{\Gamma, C, M \vdash \{\bar{\mathtt{e}}\} : \mathtt{Bag} < \mathtt{T} > |\bar{\tau}} \ (\mathrm{T}_c\text{-}8)$$

$$\frac{\Gamma, C, M \vdash e_0 : C_0|\bar{\tau} \qquad C_0 <: \mathtt{Throwable}}{\Gamma, C, M \vdash \mathtt{throw}\ e_0 : \mathtt{T}|\bar{\tau} \cup C_0} \ (\mathrm{T}_c\text{-}9)$$

$$\frac{\begin{array}{c}\Gamma, C, M \vdash e_t : \mathtt{T}_t|\bar{\tau}_t \\ C_e <: \mathtt{Exception} \qquad \Gamma \cup x : C_e, C, M \vdash e_c : \mathtt{T}_c|\bar{\tau}_c \qquad \bar{\tau}'_t = \{\tau \in \bar{\tau}_t \mid \tau \not<: \mathtt{C}_e\}\end{array}}{\Gamma, C, M \vdash \mathtt{try}\ \{e_t;\}\ \mathtt{catch}\ (C_e\ x)\ \{e_c;\}\ :\ \mathtt{T}_t \sqcup \mathtt{T}_c|\bar{\tau}'_t \cup \bar{\tau}_c} \ (\mathrm{T}_c\text{-}10)$$

$$\frac{\begin{array}{c}\Gamma, C, M \vdash e_0 : \mathtt{Bag} < \mathtt{Event} >|\bar{\tau} \\ (C <: \mathtt{Task} \wedge methname(M) = \mathtt{run}) \ \vee \ isasync(M)\end{array}}{\Gamma, C, M \vdash \mathtt{wait}\ e_0 : \mathtt{Event}|\bar{\tau}} \ (\mathrm{T}_c\text{-}15) \qquad \frac{\Gamma, C, M \vdash e_0 : C|\bar{\tau} \qquad C <: \mathtt{Task}}{\Gamma, C, M \vdash \mathtt{spawn}\ e_0 : C|\bar{\tau}} \ (\mathrm{T}_c\text{-}16)$$

**Fig. 13.** Expression typing rules for CTJ.

$\boxed{override(m, D, \bar{\mathtt{T}}_c \to \mathtt{T}_{rc}, \bar{\tau})}$

$$\frac{\begin{array}{c} mtype(m, D) = \bar{\mathtt{T}}_d \to \mathtt{T}_{rd}\ \mathtt{throws}\ \bar{\tau}_d \Rightarrow \\ \bar{\mathtt{T}}_c = \bar{\mathtt{T}}_d\ \wedge\ \mathtt{T}_{rc} = \mathtt{T}_{rd}\ \wedge\ \bar{\tau} \subset: \bar{\tau}_d \end{array}}{override(m, D, \bar{\mathtt{T}}_c \to \mathtt{T}_{rc}, \bar{\tau})}\ (\mathrm{T}_c\text{-}11)$$

$\boxed{\text{M OK in C}}$

$$\frac{\begin{array}{c} \{\bar{\mathtt{x}} : \bar{\mathtt{T}} | \emptyset, \mathtt{this} : C | \emptyset\}, C, M \vdash e_0 : \mathtt{T}_e | \bar{\tau}_e \\ \mathtt{T}_e <: \mathtt{T}_r \qquad \bar{\tau}_e \subset: \bar{\tau} \qquad CT(C) = \mathtt{class}\ C\ \mathtt{extends}\ D\ \{...\} \\ override(m, D, \bar{\mathtt{T}} \to \mathtt{T}_r, \bar{\tau}) \qquad \neg isasync(m, D) \end{array}}{\mathtt{T}_r\ m(\bar{\mathtt{T}}\ \bar{\mathtt{x}})\ \mathtt{throws}\ \bar{\tau}\ \{\mathtt{return}\ e_0;\}\ \text{OK in C}}\ (\mathrm{T}_c\text{-}12\mathrm{A})$$

$$\frac{\begin{array}{c} \{\bar{\mathtt{x}} : \bar{\mathtt{T}}, \mathtt{this} : C\}, C, M \vdash e_0 : \mathtt{T}_e | \bar{\tau}_e \qquad \mathtt{T}_e <: \mathtt{T}_r \qquad \bar{\tau}_e \subset: \bar{\tau} \\ CT(C) = \mathtt{class}\ C\ \mathtt{extends}\ D\ \{...\} \qquad override(m, D, \bar{\mathtt{T}} \to \mathtt{T}_r, \bar{\tau}) \\ isasync(m, D) \qquad \neg(C <: \mathtt{Task}\ \wedge\ m = \mathtt{run}) \end{array}}{\mathtt{async}\ \mathtt{T}_r\ m(\bar{\mathtt{T}}\ \bar{\mathtt{x}})\ \mathtt{throws}\ \bar{\tau}\ \{\mathtt{return}\ e_0;\}\ \text{OK in C}}\ (\mathrm{T}_c\text{-}12\mathrm{B})$$

$\boxed{\text{C OK}}$

$$\frac{\begin{array}{c} K = C(\bar{\mathtt{T}}_s\ \bar{\mathtt{f}}_s, \bar{\mathtt{T}}_c\ \bar{\mathtt{f}}_c)\{\mathtt{super}(\bar{\mathtt{f}}_s); \mathtt{this}.\bar{\mathtt{f}}_c = \bar{\mathtt{f}}_c;\} \\ fields(C_s) = \bar{\mathtt{T}}_s\ \bar{\mathtt{f}}_s \qquad \overline{M}\ \text{OK in C} \end{array}}{\mathtt{class}\ C\ \mathtt{extends}\ C_s\ \{\bar{\mathtt{T}}_c\ \bar{\mathtt{f}}_c;\ K\ \overline{M}\}\ \text{OK}}\ (\mathrm{T}_c\text{-}13)$$

$\boxed{\text{P OK}}$ $\qquad$ $\boxed{\mathcal{B}\ \text{OK}}$

$$\frac{\vdash e : \mathtt{T} | \emptyset \qquad \overline{CL}\ \text{OK}}{\vdash^c \overline{CL}\ \mathtt{return}\ e\ \text{OK}}\ (\mathrm{T}_c\text{-}14) \qquad \frac{\begin{array}{c} \vdash \{\bar{\mathtt{v}}\} : \mathtt{Bag} < \mathtt{Event} > \\ \vdash \bar{E}[\mathtt{new}\ \mathtt{Event}()] : \bar{\mathtt{T}} \end{array}}{\vdash \{(\{\bar{\mathtt{v}}\}, \overline{E[]})\}\ \text{OK}}\ (\mathrm{T}_c\text{-}17)$$

**Fig. 14.** Method, class, program, and blocked task typing rules for CTJ.

**Lemma 1 (Normal forms).** *If an (CTJ or EJ) expression $e$ is in normal form, either no reduction of the expression $e$ by the $\Longrightarrow$ relation is possible, or the reduction step must be an observable action.*

*Proof.* Immediate from the structure of each normal form and the $\Longrightarrow$ relation.

**Subject Reduction** We are now ready to relate the evaluation relation to the typing rules. Subject reduction states that, if a CTJ program in a well-typed state takes an evaluation step, the resulting program state is well-typed as well. We start first with a theorem for the $\longrightarrow$ relation and then extend this to the $\Longrightarrow_c$ relation.

**Theorem 1 ($\longrightarrow_c$ Subject Reduction).** *If $\Gamma \vdash e : \mathtt{T}_e|\bar{\tau}_e$ and $e \longrightarrow_c e'$, then $\Gamma \vdash e' : \mathtt{T}_{e'}|\bar{\tau}_{e'}$ for some $\mathtt{T}_{e'}|\bar{\tau}_{e'} <: \mathtt{T}_e|\bar{\tau}_e$.*

The proof of this theorem is based on several technical lemmas. When the given lemma is standard for soundness proofs, we omit the proof of the lemma for brevity.

**Lemma 2.** *If $mtype(m, C) = \bar{\mathtt{T}} \rightarrow \mathtt{T}_r$ `throws` $\bar{\tau}$, then $mtype(m, C') = \bar{\mathtt{T}} \rightarrow \mathtt{T}_r$ `throws` $\bar{\tau}'$, where $\bar{\tau}' \subseteq: \bar{\tau}$, for all $C' <: C$.*

**Lemma 3 (Non-exception values).** *Non-exception values of the form $v ::=$ `new` $C()$ | `new` $C(\bar{\mathtt{v}})$ | $\{\bar{\mathtt{v}}\}$ have a type of the form $\mathtt{T}|\emptyset$.*

**Lemma 4 (Term substitution preserves typing).** *If $\Gamma \cup \bar{\mathtt{x}} : \bar{\mathtt{T}} \vdash e_0 : \mathtt{T}_0|\bar{\tau}_0$, and $\Gamma \vdash \bar{\mathtt{v}} : \bar{\mathtt{T}}'|\emptyset$ where $\bar{\mathtt{T}}' <: \bar{\mathtt{T}}$, then $\Gamma \vdash [\bar{\mathtt{v}}/\bar{\mathtt{x}}]e_0 : \mathtt{T}'_0|\bar{\tau}_0$ for some $\mathtt{T}'_0 <: \mathtt{T}_0$.*

**Lemma 5 (Weakening).** *If $\Gamma \vdash e : \mathtt{T}|\bar{\tau}$, then $\Gamma \cup x : \mathtt{T}' \vdash e : \mathtt{T}|\bar{\tau}$.*

**Lemma 6 (Return types).** *If $mtype(m, C) = \bar{\mathtt{T}} \rightarrow \mathtt{T}_r$ `throws` $\bar{\tau}$ and $mbody(m, C) = (\bar{\mathtt{x}}, e)$, then for some $C'$ where $C <: C'$, there exists some $\mathtt{T}'_r$ and $\bar{\tau}'$ such that $\mathtt{T}'_r <: \mathtt{T}_r$, $\bar{\tau} \subseteq: \bar{\tau}'$, and $\Gamma \cup \bar{\mathtt{x}} : \bar{\mathtt{T}} \cup$ `this` $: C' \vdash e : \mathtt{T}'_r|\bar{\tau}'$.*

**Lemma 7 (Subtyping of joined types).** $\mathtt{T} <: (\mathtt{T} \sqcup \mathtt{T}')$ *for all types* $\mathtt{T}, \mathtt{T}'$.

**Lemma 8 (Subsets and the $\subseteq:$ relation).** *If $\bar{\tau} \subseteq \bar{\tau}'$, then $\bar{\tau} \subseteq: \bar{\tau}'$.*

*Proof (for Theorem 1).* By induction on the derivation of $e \longrightarrow e'$, using a case analysis on the evaluation rules.

We use $\mathtt{T}_e|\bar{\tau}_e$ to represent the type of expression $e$ and $\mathtt{T}_{e'}|\bar{\tau}_{e'}$ to represent the type of $e'$.

Computation rules:

- Rule E-1 (field computation): We have $e =$ (`new` $C(\bar{\mathtt{v}})).f_i$ and $e' = v_i$. By the premise of the theorem, we know that $e$ is well typed. The last rule in the type derivation of $e$ must be rule T-2, yielding a type of the form $\mathtt{T}_i|\bar{\tau}$ where $\mathtt{T}_i$ is the type of field $f_i$ in $\bar{\mathtt{T}}\ \bar{\mathtt{f}}$.

If $e$ is well-typed, then the receiver must also be well-typed. If we apply type rule T-4 and lemma 3 to $\texttt{new}\ C(\bar{\texttt{v}})$, we obtain the type $C|\emptyset$. By the premise of rule T-4, we know that each individual value in $\bar{\texttt{v}}$ must be well-typed and is a subtype of the associated field $f_i$ of class $C$. Thus, $v_i$ has type $\texttt{T}_e|\emptyset$, where $\texttt{T}_e <: \texttt{T}_i$. This is a subtype of $\texttt{T}_i|\emptyset$ – the theorem holds for this case.

– Rule E-2 (method invocation): We have $e = (\texttt{new}\ C(\bar{\texttt{v}})).m(\bar{\texttt{v}}_a)$ and $e' = [\bar{\texttt{v}}_a/\bar{\texttt{x}}, \texttt{new}\ C(\bar{\texttt{v}}_0)/\texttt{this}]e_0$. From the premise of the theorem, we know that $e$ is well typed. The last rule in the type derivation of $e$ must be rule T-3, yielding a type of the form $\texttt{T}_r|\bar{\tau}_0 \cup \bar{\tau}_m \cup \bar{\tau}_a$. From lemma 3, we know that $\bar{\tau}_0$ and $\bar{\tau}_a$ must both be $\emptyset$, resulting in a simplified type for $e$ of $\texttt{T}_r|\bar{\tau}_m$.
If $e$ is well-typed, then the subexpressions $\texttt{new}\ C(\bar{\texttt{v}})$ and $\bar{\texttt{v}}_a$ must also be well-typed. From type rule T-3 and lemma 3, we obtain a type for $\bar{\texttt{v}}_a$ of $\bar{\texttt{T}}_e|\emptyset$, where $\bar{\texttt{T}}_e <: \bar{\texttt{T}}_f$ (where $\bar{\texttt{x}} : \bar{\texttt{T}}_f$). From type rule T-4 and lemma 3, we obtain a type for $\texttt{new}\ C(\bar{\texttt{v}})$ of the form $C|\emptyset$ and a restriction that $\vdash \bar{\texttt{v}} : \bar{\texttt{T}}_e <: \bar{\texttt{T}}_c$, where $\bar{\texttt{T}}_c$ represents the types of class $C$'s fields.
From lemma 6, we know that the subexpression $e_0$ in $e'$ has a type $\texttt{T}'_r|\bar{\tau}'$, where $\texttt{T}'_r <: \texttt{T}_r$ and $\bar{\tau}' \subseteq: \bar{\tau}_m$. Thus, the type of $e_0$ is a subtype of $\texttt{T}_r|\bar{\tau}_m$. From type rule T-12, we know that this, when used in $e_0$, has a type of $C'|\emptyset$, where $C'$ is the class where the method $m$ is defined. Based on rule T-11, we know that $C <: C'$.
We can now derive a type for $e'$. $e_0$ has a type $\texttt{T}'_r|\bar{\tau}'$. We have shown above that $\bar{\texttt{v}}_a$ is a subtype of $\bar{\texttt{x}}$ and that $\texttt{new}\ C(\bar{\texttt{v}})$ has a type of $C|\emptyset$, which is a subtype of this in $e_0$. Thus, when we substitute $\bar{\texttt{v}}_a$ for $\bar{\texttt{x}}$ and $\texttt{new}\ C(\bar{\texttt{v}})$ for this in $e_0$, lemma 4 tells us that we obtain an expression $e'$ whose type is $\texttt{T}''_r|\bar{\tau}''$, where $\texttt{T}''_r|\bar{\tau}'' <: \texttt{T}'_r|\bar{\tau}'$ (the type of $e_0$).
Since we have also shown that $\texttt{T}'_r <: \texttt{T}_r$ and $\bar{\tau}' \subseteq: \bar{\tau}_m$. By subtyping rule S-Trans, this implies that $e'$ is a subtype of $e$: $\texttt{T}''_r|\bar{\tau}'' <: \texttt{T}_r|\bar{\tau}_m$.

– Rule E-3 (casts of classes): We have $e = (C')\texttt{new}\ C(\bar{\texttt{v}})$ and $e' = \texttt{new}\ C(\bar{\texttt{v}})$. By the premise of the theorem, we know that $e$ is well-typed. Either typing rules T-5, T-6, or T-7could apply, depending on the relationship between $C'$ and the type of subexpression $e_0 = \texttt{new}\ C(\bar{\texttt{v}})$ (we will call this type $\texttt{T}_0$). Since $e_0$ is a subexpression of $e$, it must be well-typed. Rule T-4 must be the last typing rule in the derivation. From the consequence of rule T-4 and lemma 3, we have $\vdash \texttt{new}\ C(\bar{\texttt{v}}) : C|\emptyset$ and $\texttt{T}_0 = C$. Since the premise of the evaluation rule gives us $C <: C'$, we have $\texttt{T}_0 <: C'$ and thus the last typing rule for the derivation of $e$ must be T-5. From our type derivation of $\texttt{new}\ C(\bar{\texttt{v}})$ above, we have the premise $\vdash e_0 : C|\emptyset$. Substituting this into the consequence of rule T-5, we get $\texttt{T}_e|\bar{\tau}_e = C'|\emptyset$.
Since $e' = e_0$, we have already derived a type for $e'$ using type rule T-4: $C|\emptyset$. From the premise of the evaluation rule, we have $C <: C'$, so $\texttt{T}_{e'}|\bar{\tau}_{e'} <: \texttt{T}_e|\bar{\tau}_e$.

– Rule E-4 (casts of bags): We have $e = (\texttt{Bag} < \texttt{T} >)\{\bar{\texttt{v}}\}$ and $e' = \{\bar{\texttt{v}}\}$. We must find a type derivation for $e$ consistent with the premise $\forall \texttt{T}_i \in \bar{\texttt{T}}.\texttt{T}_i <: \texttt{T}$, where $\texttt{T}_i$ is the type of the associated element of the sequence $\bar{\texttt{v}}$, $v_i$. The subexpression $e_0 = \{\bar{\texttt{v}}\}$ can be typed by rule T-8. To ensure that $e$ is well-typed, the premises of this rule and lemma 3 give us the restrictions $\texttt{T}' = \sqcup \texttt{T}_i$ and $\bar{\tau} = \emptyset$, where $\vdash v_i : \texttt{T}_i|\emptyset$. Substituting these restrictions in the

consequence for rule T-8, we get a type for $e_0$ of $\mathtt{T}' > | \emptyset$. Since $\mathtt{T}'$ is the join of all element types $\mathtt{T}_i$ and all are subtypes of $\mathtt{T}$, $\mathtt{T}'$ must be a subtype of $\mathtt{T}$. Thus, we can apply type rule T-5to $e$. If we substitute in the restrictions obtained from the type derivation of $\{\bar{\mathtt{v}}\}$ into the consequence of rule T-5, we get $\mathtt{T}_e|\bar{\tau}_e = \mathtt{Bag} < \mathtt{T} > | \emptyset$.

From the above analysis, we already have a type derivation for $e'$ using rule T-8: $\vdash \{\bar{\mathtt{v}}\} : \mathtt{Bag} < \mathtt{T}' > | \emptyset$. Since $\mathtt{T}' <: \mathtt{T}$, we have $\mathtt{T}_{e'}|\bar{\tau}_{e'} <: \mathtt{T}_e|\bar{\tau}_e$.

- Rule E-5 (throw within a constructor): We have $e = \mathtt{new}\ C(\bar{\mathtt{v}}, \mathtt{throw}\ v_e, \bar{\mathtt{e}})$ and $e' = \mathtt{throw}\ v_e$. From the form of $e$, we know that the last type rule in its derivation must be rule T-4. In order for $e$ to be well-typed, the premises of rule T-4 must hold. This gives us the restriction $\vdash \bar{\mathtt{v}}, \mathtt{throw}\ v_e, \bar{\mathtt{e}} : \bar{\mathtt{T}}_e|\bar{\tau}$, where $\bar{\mathtt{T}}_e <: \bar{\mathtt{T}}_f$ and $\bar{\mathtt{T}}_f$ represents the types of class $C$'s fields. From lemma 3, we know that $\bar{\mathtt{v}}$ cannot throw any exceptions. We will represent the possible exceptions thrown by $\bar{\mathtt{e}}$ as $\bar{\tau}_{\bar{e}}$.

  Based on the restrictions from the premises of T-4, the type of $\mathtt{throw}\ v_e$ must be a subtype of the corresponding field's type, $\mathtt{T}_i$. Type rule T-9 is the only rule which matches the form of $\mathtt{throw}\ v_e$, so this must be the last rule in its type derivation. Since the type $\mathtt{T}$ in the consequence is unbound, we select $\mathtt{T} = \mathtt{T}_i$ to satisfy the restriction derived above from rule T-4. From the premises of rule T-9, we obtain the restrictions $\vdash v_e : C_0|\bar{\tau}_0$ and $C_0 <: \mathtt{throwable}$. Since $v_e$ is a non-exception value, we can use lemma 3 to simplify $\bar{\tau}_0$ to $\emptyset$. Substituting these restrictions into the consequence of rule T-9, we get $\vdash \mathtt{throw}\ v_e : \mathtt{T}_i|C_0$. We can now substitute the types we have computed for $\bar{\mathtt{v}}, \mathtt{throw}\ v_e, \bar{\mathtt{e}}$ into the consequence of rule T-4 to obtain a type for $e$: $\mathtt{T}_e|\bar{\tau}_e = C|\bar{\tau}_{\bar{e}} \cup C_0$.

  We now derive a type for $e'$. Since $e' = \mathtt{throw}\ v_e$, $v_e$ must have the same type as the corresponding subexpression in $e$: $C_0|\emptyset$. Based on the form of $e'$, the last type rule in its type derivation must be rule T-9. We have established above that $v_e$ satisfies the premises of this rule. Since the type $\mathtt{T}$ is unbound, we now select $\mathtt{T} = C$. Substituting this into the consequence for rule T-9, we get $\vdash e' : C|C_0$. By lemma 8, $C_0 \subseteq: \bar{\tau}_{\bar{e}} \cup C_0$. Thus, we have $\mathtt{T}_{e'}|\bar{\tau}_{e'} <: \mathtt{T}_e|\bar{\tau}_e$.
- Rules E-6 – E-11, $E_c$-24, and $E_c$-26: Similar reasoning as that for rule E-5 can be applied to these rules. In each case, $e$ is a well-typed expression containing the subexpression $\mathtt{throw\ new}\ C_e(\bar{\mathtt{v}})$ and $e' = \mathtt{throw\ new}\ C_e(\bar{\mathtt{v}})$. From the restrictions we obtain by the type derivation of $e$ and type rule T-9, the type of $e'$ is $\mathtt{T}|C$. Since we can choose $\mathtt{T} = \mathtt{T}_e$ and $\bar{\tau}_e$ must contain $C_e$ ($e'$ is a subexpression of $e$ and not within a $\mathtt{try\text{-}catch}$ block), $\mathtt{T}_{e'}|\bar{\tau}_{e'}$ must be a subtype of $\mathtt{T}_e|\bar{\tau}_e$.
- Rule E-12 (try of a non-exception value): We have $e = \mathtt{try}\ \{v;\}\ \mathtt{catch}\ (C_e\ x)\ \{e_c;\}$ and $e' = v$. From the premise, we know that $e$ is well-typed. Based on the form of $e$, type rule T-10 must be the last rule used in the type derivation. Substituting the subexpressions of $e$ into the premises of rule T-10, we get $\vdash v : \mathtt{T}_t|\bar{\tau}_t$, $x{:}C_e \vdash e_c : \mathtt{T}_c|\bar{\tau}_c$, and $\bar{\tau}'_t = \{\tau \in \bar{\tau}_t \mid \tau \not<: \mathtt{C}_e\}$. From lemma 3, we know that $\bar{\tau}_t = \emptyset$ and thus $\bar{\tau}'_t = \emptyset$. Substituting these types into the consequence of rule T-10, we get $\mathtt{T}_e|\bar{\tau}_e = \mathtt{T}_t \sqcup \mathtt{T}_c|\bar{\tau}_c$.

Since $e' = v$ is the same $v$ we typed as a subexpression of $e$, $e'$ must have the same type as $v$. Thus, $\mathtt{T}_{e'}|\bar{\tau}_{e'} = \mathtt{T}_t|\emptyset$. When joining a type with other types, the resulting type must be equal to or a supertype of the original type. Thus, $\mathtt{T}_t <: \mathtt{T}_t \sqcup \bar{\mathtt{T}}_c$. This fact and $\emptyset \subseteq: \bar{\tau}_c$, lead to $\mathtt{T}_t \sqcup \mathtt{T}_c|\bar{\tau}_c <: \mathtt{T}_t|\emptyset$. Thus, $\mathtt{T}_e|\bar{\tau}_e <: \mathtt{T}_{e'}|\bar{\tau}_{e'}$.

– Rule E-13 (catch of an exception): We have $e = \texttt{try}\ \{\texttt{throw new}\ C(\bar{\mathtt{v}});\}\ \texttt{catch}\ (C_e\ x)\ \{e_c;\}$ and $e' = [v/x]e_c$. From the premise, we know that $e$ is well-typed. Based on the form of $e$, type rule T-10 must be the last rule used in the type derivation. Substituting the subexpressions of $e$ into the premises of rule T-10, we get $\vdash \texttt{throw new}\ C(\bar{\mathtt{v}}) : \mathtt{T}_t|\bar{\tau}_t$, $x : C_e \vdash e_c : \mathtt{T}_c|\bar{\tau}_c$, and $\bar{\tau}'_t = \{\tau \in \bar{\tau}_t \mid \tau \not<: \mathtt{C}_e\}$.

  We wish to further narrow the type of $e_t = \texttt{throw new}\ C(\bar{\mathtt{v}})$, so we construct a type derivation. Based on the form of $e_t$, the last rule in the derivation must be type rule T-9. By rule T-4 and lemma 3, the subexpression $\texttt{new}\ C(\bar{\mathtt{v}})$ can be typed as $C|\emptyset$. From the premises of rule T-9, we obtain the restriction $C <: \texttt{Throwable}$. Substituting the type of $\texttt{new}\ C(\bar{\mathtt{v}})$ into the consequence of rule T-9, and choosing $\mathtt{T}_c$ for the unbound type variable in the consequence, we obtain $\vdash e_t : \mathtt{T}_c|C$. Relating this to $e$ and the premises of rule T-10, we get $\mathtt{T}_t = \mathtt{T}_c$ and $\bar{\tau}_t = C$.

  From the premise of evaluation rule E-13 that $C <: C_0$ and $\bar{\tau}_t = C$, we find that $\bar{\tau}'_t = \emptyset$. Intuitively, no exceptions are thrown by the `try` block which are not caught by the catch block. We can now substitute what we have derived into the consequence of type rule T-10 to obtain a type for $e$: $\mathtt{T}_e|\bar{\tau}_e = \mathtt{T}_c|\bar{\tau}_c$.

  We now consider the type of $e'$. In the type derivation of $e$, we gave $e_c$ (the catch block's body) the type $\mathtt{T}_c|\bar{\tau}_c$. This type was computed in the type environment $\Gamma = x : C_e$. The exception value $v$ thrown from the `try` block has the form $\texttt{new}\ C(\bar{\mathtt{v}})$, which we typed using rule T-4 as $C|\emptyset$. From the premise of the evaluation rule, we have $C <: C_0$. Thus, the type of $v$ is a subtype of the type of $x$. From lemma 4, we get $\vdash [v/x]e_c : \mathtt{T}'_c|\bar{\tau}'_c$, where $\mathtt{T}'_c <: \mathtt{T}_c$ and $\bar{\tau}'_c \subseteq: \bar{\tau}_c$.

  We now have $\mathtt{T}_e|\bar{\tau}_e = \mathtt{T}_c|\bar{\tau}_c$ and $\mathtt{T}_{e'}|\bar{\tau}_{e'} = \mathtt{T}'_c|\bar{\tau}'_c$. Thus, $\mathtt{T}_{e'}|\bar{\tau}_{e'} <: \mathtt{T}_e|\bar{\tau}_e$.

– Rule E-14 (exception escaping a try-catch block): We have $e = \texttt{try}\ \{\texttt{throw new}\ C(\bar{\mathtt{v}})\}\ \texttt{catch}\ (C_e\ x)\ \{e_c;\}$ and $e' = \texttt{throw new}\ C(\bar{\mathtt{v}})$. From the premise, we know that $e$ is well-typed. Based on the form of $e$, type rule T-10 must be the last rule used in the type derivation. Substituting the subexpressions of $e$ into the premises of rule T-10, we get $\vdash \texttt{throw new}\ C(\bar{\mathtt{v}}) : \mathtt{T}_t|\bar{\tau}_t$, $x : C_e \vdash e_c : \mathtt{T}_c|\bar{\tau}_c$, and $\bar{\tau}'_t = \{\tau \in \bar{\tau}_t \mid \tau \not<: \mathtt{C}_e\}$.

  We wish to further narrow the type of $e_t = \texttt{throw new}\ C(\bar{\mathtt{v}})$, so we construct a type derivation. Based on the form of $e_t$, the last rule in the derivation must be type rule T-9. By rule T-4 and lemma 3, the subexpression $\texttt{new}\ C(\bar{\mathtt{v}})$ can be typed as $C|\emptyset$. From the premises of rule T-9, we obtain the restriction $C <: \texttt{Throwable}$. Substituting the type of $\texttt{new}\ C(\bar{\mathtt{v}})$ into the consequence of rule T-9and choosing $\mathtt{T}_c$ for the unbound type variable $\mathtt{T}$, we obtain $\vdash e_t : \mathtt{T}_c|C$.

From the premise of evaluation rule E-14 that $C \not<: C_e$ and from $\bar{\tau}_t = C$, we find that $\bar{\tau}'_t$ must be $C$. We can now substitute what we have derived into the consequence of type rule T-10 to obtain a type for $e$: $\mathtt{T}_e|\bar{\tau}_e = \mathtt{T}_c|C \cup \bar{\tau}_c$. We now consider the type of $e'$. Since $e'$ is just $e_t$, we can reuse the type derivation for $e_t$ above: $\vdash e_t : \mathtt{T}_c|C$. Since $C \subseteq: C \cup \bar{\tau}_c$, we have $\mathtt{T}_{e'}|\bar{\tau}_{e'} <: \mathtt{T}_e|\bar{\tau}_e$.

Congruence rules:

- Rule E-15 (field receiver reduction): We have $e = e_0.f$ and $e' = e'_0.f$.
  By the theorem's premise, the expression $e$ and subexpression $e_0$ must be well typed. The last rule in the type derivation of $e$ must be rule T-2, yielding a type of the form $\mathtt{T}_i|\bar{\tau}$ where $\mathtt{T}_i$ is the type of field $f_i$ in $\bar{\mathtt{T}}\ \bar{\mathtt{f}}$. By the premise of rule T-2, the subexpression $e_0$ has the type $C_0|\bar{\tau}$. By the inductive hypothesis, we know that, if $e_0 \rightarrow e'_0$ and $\vdash e'_0 : C'_0|\bar{\tau}'$, then $C'_0 <: C_0$ and $\bar{\tau}' \subseteq: \bar{\tau}$. By subtyping rule S-Cls, if $C'_0 <: C_0$, then field $f_i$ of $C_0$ is also present in $C'_0$ and will have the same type, $\mathtt{T}_i$. In other words, the fields of $C'_0$ are a superset of those in $C_0$: $fields(C'_0) = \bar{\mathtt{T}} \cup \bar{\mathtt{T}}'\ \bar{\mathtt{f}} \cup \bar{\mathtt{f}}'$.
  We can now derive a type for $e'$ using type rule T-2. From premises $\vdash e'_0 : C'_0|\bar{\tau}$ and $fields(C'_0) = \bar{\mathtt{T}} \cup \bar{\mathtt{T}}'\ \bar{\mathtt{f}} \cup \bar{\mathtt{f}}'$, we obtain $\vdash e'_0.f_i : \mathtt{T}_i|\bar{\tau}'$. Since $\bar{\tau}' \subseteq: \bar{\tau}$, we have $\mathtt{T}_{e'}|\bar{\tau}_{e'} <: \mathtt{T}_e|\bar{\tau}_e$.
- Rule E-16 (method receiver reduction): We have $e = e_0.m(\bar{\mathtt{e}})$ and $e' = e'_0.m(\bar{\mathtt{e}})$. By the theorem's premise, the expressions $e$ and subexpressions $e_0$ and $\bar{\mathtt{e}}$ are well-typed. Based on the form of $e$ , the last rule in the type derivation for $e$ must be type rule T-3. The premises of this rule give us the restrictions $\vdash e_0 : C_0|\bar{\tau}_0$, $mtype(m, C_0) = \bar{\mathtt{T}}_f \rightarrow \mathtt{T}_r\ \mathtt{throws}\ \bar{\tau}_m$, $\vdash \bar{\mathtt{e}} : \bar{\mathtt{T}}_a|\bar{\tau}_a$, and $\bar{\mathtt{T}}_a <: \bar{\mathtt{T}}_f$. From the consequence of this rule, the type of $e$ must have the form $\mathtt{T}_r|\bar{\tau}_0 \cup \bar{\tau}_m \cup \bar{\tau}_a$.
  We now derive a type for $e'$. If $e'$ is well-typed, then, based on the form of $e'$, the last type rule in the derivation must be rule T-3. By the inductive hypothesis, we know that, if $e_0 \rightarrow e'_0$, then $e'_0$ must be a subtype of $e_0$. From the derivation above we have $\vdash e_0 : C_0|\bar{\tau}_0$. Thus, $e'_0$ must have a type of the form $C'_0|\bar{\tau}'_0$ where $C'_0 <: C_0$ and $\bar{\tau}'_0 \subseteq: \bar{\tau}_0$. This gives use the first premise of rule T-3. To derive the second premise of rule T-3, we use lemma 2 and the fact that $C'_0 <: C_0$: $mtype(m, C'_0) = \bar{\mathtt{T}}_f \rightarrow \mathtt{T}_r\ \mathtt{throws}\ \bar{\tau}'_m$, where $\bar{\tau}'_m \subseteq: \bar{\tau}_m$. Since the subexpression $\bar{\mathtt{e}}$ is the same in $e$ and $e'$, then it must have the same type in $e'$: $\vdash \bar{\mathtt{e}} : \bar{\mathtt{T}}_a|\bar{\tau}_a$. This gives us the third and fourth premises for rule 3. Now we can substitute the types we've derived for each subexpression into the consequence for rule 3: $\vdash e' : \mathtt{T}_r|\bar{\tau}'_0 \cup \bar{\tau}'_m \cup \bar{\tau}_a$. Since $\bar{\tau}'_0 \subseteq: \bar{\tau}_0$ and $\bar{\tau}'_m \subseteq: \bar{\tau}_m$, then $\mathtt{T}_{e'}|\bar{\tau}_{e'} <: \mathtt{T}_e|\bar{\tau}_e$.
- Rule E-17 (method argument reduction): We have $e = v_0.m(\bar{\mathtt{v}}, e_i, \bar{\mathtt{e}})$ and $e' = v_0.m(\bar{\mathtt{v}}, e'_i, \bar{\mathtt{e}})$. By the theorem's premise, the expressions $e$ and subexpressions $v$ and $\bar{\mathtt{v}}, e_i, \bar{\mathtt{e}}$ are well-typed. Based on the form of $e$, the last rule in the type derivation for $e$ must be type rule T-3. The premises of this rule give us the restrictions $\vdash v : C_0|\bar{\tau}_0$, $mtype(m, C_0) = \bar{\mathtt{T}}_f \rightarrow \mathtt{T}_r\ \mathtt{throws}\ \bar{\tau}_m$, $\vdash \bar{\mathtt{v}} : \bar{\mathtt{T}}_v|\bar{\tau}_v$, $\vdash e_i : \mathtt{T}_i|\bar{\tau}_i$, and $\vdash \bar{\mathtt{e}} : \bar{\mathtt{T}}_a|\bar{\tau}_a$, where $\bar{\mathtt{T}}_v, \mathtt{T}_i, \bar{\mathtt{T}}_a <: \bar{\mathtt{T}}_f$. By lemma 3, we know that $\bar{\tau}_0 = \emptyset$ and $\bar{\tau}_v = \emptyset$. Substituting these restrictions into the consequence of rule T-3, we get $\mathtt{T}_e|\bar{\tau}_e = \mathtt{T}_r|\bar{\tau}_m \cup \bar{\tau}_i \cup \bar{\tau}_a$.

We now derive a type for $e'$. Based on the form of $e'$, the last type rule in the derivation must be rule T-3. Since the receiver of the method call is the same for both $e$ and $e'$, the types of the receiver and the method must be the same as well. This gives us the first two premises of rule T-3: $\vdash v : C_0|\emptyset$ and $mtype(m, C_0) = \bar{\mathtt{T}}_f \rightarrow \mathtt{T}_r$ `throws` $\bar{\tau}_m$. Likewise, the subexpressions $\bar{\mathtt{v}}$ and $\bar{\mathtt{e}}$ are the same in both expressions, yielding the same types: $\vdash \bar{\mathtt{v}} : \bar{\mathtt{T}}_v : \emptyset$ and $\vdash \bar{\mathtt{e}} : \bar{\mathtt{T}}_a|\bar{\tau}_a$. By the inductive hypothesis, we know that, if $e_i \rightarrow e_i'$, then $e_i'$ must be a subtype of $e_i$. Thus, $e_i'$ has a type of the form $\mathtt{T}_i'|\bar{\tau}_i'$, where $\mathtt{T}_i' <: \mathtt{T}_i$ and $\bar{\tau}_i' \subseteq: \bar{\tau}_i$. Since $\bar{\mathtt{T}}_i$ is a subtype of the associated method argument type in $\bar{\mathtt{T}}_f$ and $\bar{\mathtt{T}}_i' <: \bar{\mathtt{T}}_i$, we satisfy the fourth premise: $\bar{\mathtt{T}}_v, \mathtt{T}_i', \bar{\mathtt{T}}_a <: \bar{\mathtt{T}}_f$. Given these premises for rule T-3, the consequence gives us the following type for $e'$: $\mathtt{T}_{e'}|\bar{\tau}_{e'} = \mathtt{T}_r|\bar{\tau}_m \cup \bar{\tau}_i' \cup \bar{\tau}_a$.

Since $\bar{\tau}_i' \subseteq: \bar{\tau}_i$, then $\mathtt{T}_{e'}|\bar{\tau}_{e'} <: \mathtt{T}_e|\bar{\tau}_e$.

- Rules E-18, E-20, E-21, E-22, $E_c$-23, and $E_c$-25: These follow similar reasoning to that of rule E-17. By the premise of the theorem, $e$ is well typed. From the form of $e$ , we know that the last rule in the type derivation must be rule T-4, T-8, T-9, T-10, $\mathrm{T}_c$-15, and $\mathrm{T}_c$-16, respectively. We use the premises of these rules to derive types for the subexpressions of $e$. Then, we derive a type for $e'$. All but one subexpression is the same for both $e$ and $e'$, enabling us to use the types we derived for these subexpressions. By the premise of the evaluation rule, the remaining subexpression $e_0'$ is associated with the corresponding subexpression $e_0$ of $e$ by an evaluation step. By the inductive hypothesis, we know that $e_0'$ must be a subtype of $e_0$. This allows us to establish that the type of $e'$ is a subtype of the type of $e$.
- Rule E-19 (reduction of casts): We have $e = (\mathtt{T})e_0$ and $e' = (\mathtt{T})e_0'$. From the theorem's premise, we know that $e$ is well-typed. The last rule in the typing derivation for $e$ may be either T-5, T-6, or T-7, depending on the relationship between the type of $e_0$ and $\mathtt{T}$. Without loss of generality, we represent the type of $e_0$ as $\mathtt{T}_0|\bar{\tau}_0$.

  By the premise of the evaluation rule, $e_0 \rightarrow e_0'$. Thus, by the inductive hypothesis, we know that $e_0'$ must be a subtype of $e_0$. In other words, $e_0'$ has a type of the form $\mathtt{T}_0'|\bar{\tau}'$, where $\mathtt{T}_0' <: \mathtt{T}_0$ and $\bar{\tau}' \subseteq: \bar{\tau}$. Based on the form of $e'$, the last rule used in its type derivation must be either T-5, T-6, or T-7, depending on the relationship between $\mathtt{T}_0'$ and $\mathtt{T}$.

  We now derive types for $e$ and $e'$, based on the possible relationships $\mathtt{T}$ has with $\mathtt{T}_0$ and $\mathtt{T}_0'$:

  1. $\mathtt{T}_0 <: \mathtt{T}$: Type rule T-5 has the premise $\mathtt{T}_0 <: \mathtt{T}$ and thus must be the last rule in the type derivation for $e$. Substituting the type of $e_0$ into the consequence of this rule, we get $\mathtt{T}_e|\bar{\tau}_e = \mathtt{T}|\bar{\tau}_0$.

     Since the subtyping relationship is transitive, $\mathtt{T}_0 <: \mathtt{T}$ implies that $\mathtt{T}_0' <: \mathtt{T}$. Thus, if $e'$ is well-typed, type rule T-5 must be the last rule in its type derivation. By the inductive hypothesis, $e_0'$ is well-typed. Substituting the type we derived for $e_0'$ above into the consequence of rule T-5, we get $\mathtt{T}_{e'}|\bar{\tau}_{e'} = \mathtt{T}|\bar{\tau}_0'$. Since $\bar{\tau}_0' \subseteq: \bar{\tau}_0$, $\mathtt{T}_{e'}|\bar{\tau}_{e'} <: \mathtt{T}_e|\bar{\tau}_e$.

2. $\mathtt{T} <: \mathtt{T}_0$ and $\mathtt{T} \neq \mathtt{T}_0$: Type rule T-6 has the premises $\mathtt{T} <: \mathtt{T}_0$ and $\mathtt{T} \neq \mathtt{T}_0$ and thus must be the last rule in the type derivation for $e$. Substituting the type of $e_0$ into the consequence of this rule, we get $\mathtt{T}_e|\bar{\tau}_e = \mathtt{T}|\bar{\tau}_0$.
   Next, we derive a type for $e'$. By the inductive hypothesis, $e'_0$ is well-typed and $\mathtt{T}'_0 <: \mathtt{T}_0$. However, this relationship does not constrain the relationship between $\mathtt{T}$ and $\mathtt{T}'_0$. Thus, any of type rule T-5, T-6, or T-7may be the last rule in the type derivation of $e'$. However, substituting the type of $e_0$ into the consequences of each of these rules yields the same type: $\mathtt{T}_{e'}|\bar{\tau}_{e'} = \mathtt{T}|\bar{\tau}'_0$. Since $\bar{\tau}'_0 \subseteq: \bar{\tau}_0$, we get $\mathtt{T}_{e'}|\bar{\tau}_{e'} <: \mathtt{T}_e|\bar{\tau}_e$.
3. $\mathtt{T}_0 \not<: \mathtt{T}$ and $\mathtt{T} \not<: \mathtt{T}_0$: Type rule T-7 has the premises $\mathtt{T}_0 \not<: \mathtt{T}$ and $\mathtt{T} \not<: \mathtt{T}_0$ and thus must be the last rule in the type derivation for $e$. Substituting the type of $e_0$ into the consequence of this rule, we get $\mathtt{T}_e|\bar{\tau}_e = \mathtt{T}|\bar{\tau}_0$.
   By the inductive hypothesis, $e'_0$ is well-typed and $\mathtt{T}'_0 <: \mathtt{T}_0$. However, by contradiction, if $\mathtt{T}$ and $\mathtt{T}_0$ do not have a relationship, then $\mathtt{T}$ and $\mathtt{T}'_0$ do not have a relationship either. Thus, if $e'$ is well-typed, then type rule T-7 must be the last rule in its derivation. Substituting the type of $e'_0$ into the consequence for rule T-7, we get $\mathtt{T}_{e'}|\bar{\tau}_{e'} = \mathtt{T}|\bar{\tau}'_0$. Since $\bar{\tau}'_0 \subseteq: \bar{\tau}_0$, then $\mathtt{T}_{e'}|\bar{\tau}_{e'} <: \mathtt{T}_e|\bar{\tau}_e$.

We now extend subject reduction to CTJ program states.

**Theorem 2 ($\Longrightarrow_c$ Subject Reduction).** *If* $\vdash e : \mathtt{T}_e|\bar{\tau}_e$ *and* $\mathcal{B}$ *OK and* $e|\mathcal{B} \Longrightarrow_c e'|\mathcal{B}'$*, then* $\vdash e' : \mathtt{T}_{e'}|\bar{\tau}_{e'}$ *and* $\mathcal{B}'$ *OK.*

*Proof.* By using a case analysis on the evaluation rules:

- Rule $E_c$-Con (step via $\longrightarrow_c$ relation): By theorem 1, we have $\mathtt{T}_{e'}|\bar{\tau}_{e'} <: \mathtt{T}_e|\bar{\tau}_e$. The blocked task set remains unchanged and $\mathcal{B}$ OK is a premise of the theorem.
- Rule $E_c$-Throw (throw): The expression $e$ remains unchanged and, thus is well-typed. $\mathcal{B}'$ is the empty set, which is well-typed by rule $\mathrm{T}_c$-17.
- Rule $E_c$-Wait (wait): The value `new Object()` is well-typed by rule T-4. By the premise of the theorem, $E[\texttt{wait } \{\bar{\mathtt{v}}\}]$ is well-typed and, by type rule $\mathrm{T}_c$-15, the `wait` call has type $\texttt{Event}|\emptyset$. Thus, $E[\texttt{new Event}()]$ will be well-typed and $\mathcal{B}'$ OK by rule $\mathrm{T}_c$-17.
- Rules $E_c$-Run and $E_c$-$\eta_\emptyset$Run (selection of an event): By rule $\mathrm{T}_c$-17, $E[]$ is well-typed when the placeholder is replaced with an object of type `Event`. Thus, $E[\eta]$ and $E[\eta_0]$ are well-typed. $\mathcal{B}' = \mathcal{B} \setminus (\{\bar{\mathtt{v}}\}, E[])$ and is OK by rule $\mathrm{T}_c$-17.
- Rule $E_c$-Spn (spawn): By type rules T-4 and $\mathrm{T}_c$-16, $\texttt{new } C(\bar{\mathtt{v}})$ and $\texttt{spawn } C(\bar{\mathtt{v}})$ both have type $C|\emptyset$. Thus, substituting $\texttt{new } C(\bar{\mathtt{v}})$ for $\texttt{spawn } C(\bar{\mathtt{v}})$ will result in a well-typed expression. From type rule $\mathrm{T}_c$-16 we know that $C <: \texttt{Task}$. Thus, the expression $(\texttt{new } C(\bar{\mathtt{v}})).run(\texttt{new Event}())$ is well-typed by rule T-3 and $\mathcal{B}$' OK by $\mathrm{T}_c$-17.

Note that, when taking a step via the $\Longrightarrow_c$ relation, a subtype relationship no longer holds between the original expression and the new expression. If the

evaluation context is of the form $E[\texttt{wait}\{\bar{\texttt{v}}\}]$, the current expression may be replaced with one from the blocked set. This new expression need not have a subtype relationship with the previous one. As we shall see in theorem 5, this does not prevent us from making the usual claims about the type safety of CTJ programs.

**Progress** The progress theorem states that well-typed CTJ programs cannot get "stuck", except when a bad cast occurs.

We first state a technical lemma needed for the proofs:

**Lemma 9 (Canonical forms).** *The forms of values are related to their types as follows:*

- *If* $\vdash \nu : C|\bar{\tau}$*, and* $\nu$ *is a value, then* $\nu$ *has either the form* $\texttt{new}\ C(\bar{\texttt{v}})$*, where* $\vdash \bar{\texttt{v}} : \bar{\texttt{T}}_f$*, or the form* $\texttt{throw}\ \texttt{new}\ C(\bar{\texttt{v}})$*, where* $C <: \texttt{Throwable}$*.*
- *If* $\vdash \nu : \texttt{Bag} < \texttt{T} >|\bar{\tau}$*, and* $\nu$ *is a value, then* $\nu$ *has either the form* $\{\bar{\texttt{v}}\}$*, where* $\vdash \bar{\texttt{v}} : \bar{\texttt{T}}|\emptyset$ *and* $\texttt{T} = \sqcup\bar{\texttt{T}}$*, or the form* $\texttt{throw}\ \texttt{new}\ C(\bar{\texttt{v}})$*, where* $C <: \texttt{Throwable}$*.*

**Theorem 3 ($\longrightarrow_c$ Progress).** *Suppose* $\vdash e : T|\bar{\tau}$*. Then either* $e$ *is a normal form or there exists an expression* $e'$ *such that* $e \longrightarrow_c e'$*.*

*Proof.* We prove theorem 3 by induction on the depth of the derivation of $\vdash e : T|\bar{\tau}$, using a case analysis on the last type rule for each derivation

The handling of exceptions in this proof is interesting due to the type rule for `throw` (T-9) – it assigns an arbitrary type. If the exception type $\bar{\tau}$, is non-empty, we must consider values of the form $\texttt{throw}\ v_e$ whenever we consider values. Thus, expressions that may step by a computation rule may also step to a `throw` expression. This is reflected in our canonical forms lemma, which includes a `throw` expression for each value form.

We now present the detailed case analysis for each type rule:

- Rule T-1 (variables): This is a contradiction, as $x$ is not well-typed in an empty type environment.
- Rule T-2 (fields): We have $e = e_0.f_i$ where $\vdash e_0 : C_0|\bar{\tau}$. By the inductive hypothesis, one of the following is true for the subexpression $e_0$:
  - $e_0$ is a value $v$: By lemma 9, $e_0$ must have either the form $\texttt{new}\ C(\bar{\texttt{v}})$ or the form $\texttt{throw}\ \texttt{new}\ C(\bar{\texttt{v}})$. In the first case, computation rule E-1 can be applied. In the second case computation rule E-9 can be applied. Both cases yield a value for $e'$.
  - $e_0$ is one of the following normal forms: $E[\texttt{spawn}\ C(\bar{\texttt{v}})]$, $E[\texttt{wait}\ \{\bar{\texttt{v}}\}]$, $E[(T)v]$ (where $\vdash v : T'$ and $T' \not<: T$), or $v$ (a non-exception value). In this case, $e$ is also of the same form, and thus is a normal form.
  - There exists an $e_0'$ such that $e_0 \rightarrow e_0'$: Congruence rule E-15 can be applied to yield a new expression $e'$.
- Rule T-3 (method calls): We have $e = e_0.m(\bar{\texttt{e}})$ where $\vdash e_0 : C_0|\bar{\tau}_0$ and $\vdash \bar{\texttt{e}} : \bar{\texttt{T}}|\bar{\tau}_a$. By the inductive hypothesis, one of the following is true for the subexpression $e_0$:

1. $e_0$ is a value $v$. By lemma 9, $e_0$ must have either the form $\texttt{new}\ C(\bar{\texttt{v}})$ or the form $\texttt{throw new}\ C(\bar{\texttt{v}})$. In the second case, evaluation rule E-6 can be applied, yielding a value for $e'$. If $e_0$ has the form $\texttt{new}\ C(\bar{\texttt{v}})$, then we must look at the form of the method's actuals as well. We analyze each element of the sequence $\bar{\texttt{e}}$ in order, applying the inductive hypothesis:
   (a) $e_i$ is a value and thus $\bar{\texttt{e}}$ has the form $\bar{\texttt{v}}, v_i, \bar{\texttt{e}}'$. As a value, $v_i$ must have either the form $\texttt{new}\ C(\bar{\texttt{v}}')$, $\{\bar{\texttt{v}}'\}$, or $\texttt{throw}\ v_e$. In the third case, computation rule E-7 may be applied, yielding a value for $e'$. In the first two cases, if $e_i$ is not the last element in the sequence, the inductive hypothesis can be applied to the next element in the sequence. If $e_i$ is the last element in the sequence and has either the form $\texttt{new}\ C(\bar{\texttt{v}}')$ or $\{\bar{\texttt{v}}'\}$, then, computation rule E-2 may be applied to yield a new expression $e'$.
   (b) $e_i$ is one of the following normal forms: $E[\texttt{spawn}\ C(\bar{\texttt{v}})]$, $E[\texttt{wait}\ \{\bar{\texttt{v}}\}]$, or $E[(T)v]$ (where $\vdash v : T'$ and $T' \not<: T$). Then, $\bar{\texttt{e}}$ is of the form $\bar{\texttt{v}}, \textit{normal form}, \bar{\texttt{e}}'$. As a result, $e$ is of the same form as $e_i$, and thus is a also a normal form.
   (c) There exists an $e_i'$ such that $e_i \rightarrow e_i'$. Thus, $\bar{\texttt{e}}$ is of the form $\bar{\texttt{v}}, e_i, \bar{\texttt{e}}'$ and congruence rule E-17 can then be applied to yield a new expression $e'$.
2. $e_0$ is one of the following normal forms: $E[\texttt{spawn}\ C(\bar{\texttt{v}})]$, $E[\texttt{wait}\ \{\bar{\texttt{v}}\}]$, or $E[(T)v]$ (where $\vdash v : T'$ and $T' \not<: T$). In this case, $e$ is also of the same form, and thus is a normal form.
3. There exists an $e_0'$ such that $e_0 \rightarrow e_0'$. Congruence rule E-16 can then be applied to yield a new expression $e'$.

– Rule T-4 (constructors): We have $e = \texttt{new}\ C_c(\bar{\texttt{e}})$ where $\vdash \bar{\texttt{e}} : \bar{\texttt{T}}_e | \bar{\tau}$. We analyze each element of the sequence $\bar{\texttt{e}}$ in order, applying the inductive hypothesis:
1. $e_i$ is a value and thus $\bar{\texttt{e}}$ has the form $\bar{\texttt{v}}, v_i, \bar{\texttt{e}}'$. As a value, $v_i$ must have either the form $\texttt{new}\ C(\bar{\texttt{v}}')$, $\{\bar{\texttt{v}}'\}$, or $\texttt{throw}\ v_e$. In the third case, computation rule E-5 may be applied, yielding a value for $e'$. In the first two cases, if $e_i$ is not the last element in the sequence, the inductive hypothesis can be applied to the next element in the sequence. If $e_i$ is the last element in the sequence and has either the form $\texttt{new}\ C(\bar{\texttt{v}}')$ or $\{\bar{\texttt{v}}'\}$, then, $e$ is a value.
2. $e_i$ is one of the following normal forms: $E[\texttt{spawn}\ C(\bar{\texttt{v}})]$, $E[\texttt{wait}\ \{\bar{\texttt{v}}\}]$, or $E[(T)v]$ (where $\vdash v : T'$ and $T' \not<: T$). Then, $\bar{\texttt{e}}$ is of the form $\bar{\texttt{v}}, \textit{normal form}, \bar{\texttt{e}}'$. As a result, $e$ is of the same form as $e_i$, and thus is a also a normal form.
3. There exists an $e_i'$ such that $e_i \rightarrow e_i'$. Thus, $\bar{\texttt{e}}$ is of the form $\bar{\texttt{v}}, e_i, \bar{\texttt{e}}'$ and congruence rule E-18 can then be applied to yield a new expression $e'$.

– Rule T-5(casts where $\texttt{T}_0 <: \texttt{T}$): We have $e = T(e_0)$ where $\vdash e_0 : \texttt{T}_0 | \bar{\tau}$. By the induction hypothesis, one of the following is true for $e_0$:
1. $e_0$ is a value. By the premise of the typing rule, we have $\texttt{T}_0 <: \texttt{T}$. As a value, $e_0$ must have either the form $\texttt{new}\ C(\bar{\texttt{v}})$, $\{\bar{\texttt{v}}\}$, or $\texttt{throw}\ v_e$. Computation rules E-3, E-4, and E-5 (respectively) can be applied to yield a new expression $e'$.

2. $e_0$ is one of the following normal forms: $E[\texttt{spawn}\ C(\bar{\texttt{v}})]$, $E[\texttt{wait}\ \{\bar{\texttt{v}}\}]$, or $E[(T)v]$ (where $\vdash v : T'$ and $T' \not<: T$). In this case, $e$ is also of the same form, and thus is a normal form.
3. There exists an $e_0'$ such that $e_0 \rightarrow e_0'$. Congruence rule E-19 can then be applied to yield a new expression $e'$.

– Rule T-6 (casts where $\texttt{T} <: \texttt{T}_0$ and $\texttt{T} \neq \texttt{T}_0$): We have $e = T(e_0)$ where $\vdash e_0 : \texttt{T}_0|\bar{\tau}$. By the premise of the typing rule, we have $\texttt{T}_0 \not<: \texttt{T}$. By the induction hypothesis, one of the following is true for $e_0$:
1. $e_0$ is a value. As a value, $e_0$ must have either the form $\texttt{new}\ C(\bar{\texttt{v}})$, $\{\bar{\texttt{v}}\}$, or $\texttt{throw new}\ C_e(\bar{\texttt{v}})$. The third case is not possible since, by type rule T-9, $\vdash \texttt{throw new}\ C_e(\bar{\texttt{v}}) : \bot|C_e$. $\bot$ is a subtype of all other types, which contradicts the premise of typing rule T-6.
If $e_0$ has either the form $\texttt{new}\ C(\bar{\texttt{v}})$ or $\{\bar{\texttt{v}}\}$, then $e$ has the form $E[(T)v]$, where the type of $v$ is not a subtype of $T$. This is a runtime casting error.
2. $e_0$ is one of the following normal forms: $E[\texttt{spawn}\ C(\bar{\texttt{v}})]$, $E[\texttt{wait}\ \{\bar{\texttt{v}}\}]$, or $E[(T)v]$ (where $\vdash v : T'$ and $T' \not<: T$). In this case, $e$ is also of the same form, and thus is a normal form.
3. There exists an $e_0'$ such that $e_0 \rightarrow e_0'$. Congruence rule E-19 can then be applied to yield a new expression $e'$.

– Rule T-7 (casts where $\texttt{T}_0 \not<: \texttt{T}$ and $\texttt{T} \not<: \texttt{T}_0$): We have $e = T(e_0)$ where $\vdash e_0 : \texttt{T}_0|\bar{\tau}$. This case is handled in the same manner as type rule T-6.

– Rule T-8 (bags): We have $e = \{\bar{\texttt{e}}\}$ where $\vdash \bar{\texttt{e}} : \texttt{T}|\bar{\tau}$. We analyze each element of the sequence $\bar{\texttt{e}}$ in order, applying the inductive hypothesis:
1. $e_i$ is a value and thus $\bar{\texttt{e}}$ has the form $\bar{\texttt{v}}, v_i, \bar{\texttt{e}}'$. As a value, $v_i$ must have either the form $\texttt{new}\ C(\bar{\texttt{v}}')$, $\{\bar{\texttt{v}}'\}$, or $\texttt{throw}\ v_e$. In the third case, computation rule E-8 may be applied, yielding a value for $e'$. In the first two cases, if $e_i$ is not the last element in the sequence, the inductive hypothesis can be applied to the next element in the sequence. If $e_i$ is the last element in the sequence and has either the form $\texttt{new}\ C(\bar{\texttt{v}}')$ or $\{\bar{\texttt{v}}'\}$, then, $e$ is a value.
2. $e_i$ is one of the following normal forms: $E[\texttt{spawn}\ C(\bar{\texttt{v}})]$, $E[\texttt{wait}\ \{\bar{\texttt{v}}\}]$, or $E[(T)v]$ (where $\vdash v : T'$ and $T' \not<: T$). Then, $\bar{\texttt{e}}$ is of the form $\bar{\texttt{v}}, \textit{normal form}, \bar{\texttt{e}}'$. As a result, $e$ is of the same form as $e_i$, and thus is a also a normal form.
3. There exists an $e_i'$ such that $e_i \rightarrow e_i'$. Thus, $\bar{\texttt{e}}$ is of the form $\bar{\texttt{v}}, e_i, \bar{\texttt{e}}'$ and congruence rule E-20 can then be applied to yield a new expression $e'$.

– Rule T-9 (throw): We have $e = \texttt{throw}\ e_0$ where $\vdash e_0 : C_0|\bar{\tau}$ and $C_0 <: \texttt{Throwable}$. By the inductive hypothesis, one of the following is true for $e_0$:
1. $e_0$ is a value. By lemma 9, $e_0$ must have the either form $\texttt{new}\ C(\bar{\texttt{v}})$ or $\texttt{throw}\ v_e$. In the first case, $e$ is then a value. In the second case, computation rule E-11 can be applied to yield a value.
2. $e_0$ is one of the following normal forms: $E[\texttt{spawn}\ C(\bar{\texttt{v}})]$, $E[\texttt{wait}\ \{\bar{\texttt{v}}\}]$, or $E[(T)v]$ (where $\vdash v : T'$ and $T' \not<: T$). In this case, $e$ is also of the same form, and thus is a normal form.
3. There exists an $e_0'$ such that $e_0 \rightarrow e_0'$. Congruence rule E-21 can then be applied to yield a new expression $e'$.

– Rule T-10(try-catch blocks): We have $e = \texttt{try}\ \{e_t;\}\ \texttt{catch}\ (C_e\ x)\ \{e_c;\}$ where $\vdash e_t : \texttt{T}_t|\bar{\tau}_t$. By the inductive hypothesis, one of the following is true for $e_0$:
  1. $e_t$ is a value. As a value, $e_t$ must have either the form $\texttt{new}\ C(\bar{\texttt{v}})$, $\{\bar{\texttt{v}}\}$, or $\texttt{throw}\ \texttt{new}\ C_e(\bar{\texttt{v}})$. For the third case, either E-13 or E-14 can be applied, depending on the relationship between $C_e$ and the exceptions caught by the `catch` block.
     If $e_t$ has either the form $\texttt{new}\ C(\bar{\texttt{v}})$ or $\{\bar{\texttt{v}}\}$, then computation rule E-12 can be applied to $e$, yielding a value.
  2. $e_t$ is one of the following normal forms: $E[\texttt{spawn}\ C(\bar{\texttt{v}})]$, $E[\texttt{wait}\ \{\bar{\texttt{v}}\}]$, or $E[(T)v]$ (where $\vdash v : T'$ and $T' \not<: T$). In this case, $e$ is also of the same form, and thus is a normal form.
  3. There exists an $e'_t$ such that $e_t \rightarrow e'_t$. Congruence rule E-22 can then be applied to yield a new expression $e'$.
– Rule $\text{T}_c$-15 (wait): We have $e = \texttt{wait}\ e_0$ where $\vdash e_0 : \texttt{Bag} < \texttt{Event} > |\bar{\tau}$. By the inductive hypothesis, one of the following is true for $e_0$:
  1. $e_0$ is a value. By lemma 9, $e_0$ must have either the form $\{\bar{\texttt{v}}\}$ or $\texttt{throw}\ \texttt{new}\ C_e(\bar{\texttt{v}})$. In the first case, we have an expression of the form $E[\texttt{wait}\ \{\bar{\texttt{v}}\}]$ where each element of $\bar{\texttt{v}}$ is a subtype of `Event`. In the second case, computation rule $E_c$-24 can be applied to yield a value.
  2. $e_0$ is an expression of the form $E[(T)v]$, where the type of $v$ is not a subtype of $T$. Then, $e$ is also of the form $E[(T)v]$.
  3. $e_0$ is one of the following normal forms: $E[\texttt{spawn}\ C(\bar{\texttt{v}})]$, $E[\texttt{wait}\ \{\bar{\texttt{v}}\}]$, or $E[(T)v]$ (where $\vdash v : T'$ and $T' \not<: T$). In this case, $e$ is also of the same form, and thus is a normal form.
  4. There exists an $e'_0$ such that $e_0 \rightarrow e'_0$. Congruence rule $E_c$-23 can then be applied to yield a new expression $e'$.

**Theorem 4 ($\Longrightarrow_c$ Progress).** *Suppose $\vdash e : T|\bar{\tau}$ and $\mathcal{B}$ OK. Then one of the following must be true:*

– *$e$ is a value and $\mathcal{B} = \emptyset$.*
– *$e$ is of the form $E[(T)v]$, where $E[]$ is an evaluation context and $v$ is a value whose type is not a subtype of $T$. We call this a runtime cast error.*
– *There exists an expression $e'$ and set of blocked tasks $\mathcal{B}'$ such that $e|\mathcal{B} \Longrightarrow_c e'|\mathcal{B}'$.*

*Proof.* From theorem 3, we know that either $e$ is a normal form or there exists an expression $e'$ such that $e \longrightarrow e'$. If $e \longrightarrow e'$, then, by evaluation rule $E_c$-Con, $e|\mathcal{B} \overset{\epsilon}{\Longrightarrow} e'|\mathcal{B}$. If $e$ is a normal form, we perform a case analysis over normal forms and the contents of $\mathcal{B}$:

– If $e$ has the form $E[\texttt{spawn}\ C(\bar{\texttt{v}})]$ or $E[\texttt{wait}\ \{\bar{\texttt{v}}\}]$, then $e$ steps by rules $E_c$-Spn and $E_c$-Wait, respectively.
– If $e$ has the form $E[(T)v]$ where $v$ is a value whose type is not a subtype of $T$, then a runtime cast error has occurred, and no step may be taken.
– If $e$ is an exception value, and $\mathcal{B}$ is not empty, then $e|\mathcal{B} \Longrightarrow e'|\mathcal{B}'$ by evaluation rule $E_c$-Throw.

– If $e$ is a non-exception value and $\mathcal{B}$ is not empty, then $e|\mathcal{B} \Longrightarrow e'|\mathcal{B}'$ by either evaluation rule $E_c$-Run or $E_c$-$\eta_\emptyset$Run.
– If $e$ is a value and $\mathcal{B} = \emptyset$, then the program has terminated and cannot step.

**Soundness** We now combine theorems 2 and 4 to extend our guarantee to entire program executions. First, we need two lemmas regarding exceptions:

**Lemma 10 ($\longrightarrow_c$ Exceptions).** *If $\vdash e : \mathtt{T}|\bar{\tau}$, and $e \longrightarrow_c^* \mathtt{throw\ new}\ C(\bar{\mathtt{v}})$, then $\exists \tau_i \in \bar{\tau} \mid C <: \tau_i$.*

*Proof.* By induction over evaluation steps. By the inductive hypothesis, an arbitrary expression from the sequence $e_i$ has type $\vdash e_i : T_i|\bar{\tau}_i$, where $\bar{\tau}_i \subseteq: \bar{\tau}$. If $e_i \longrightarrow e_{i+1}$, then, by theorem 2, $\vdash e_{i+1} : T_{i+1}|\bar{\tau}_{i+1}$, where $\bar{\tau}_{i+1} \subseteq: \bar{\tau}_i$.

From type rule T-9, $\vdash \mathtt{throw\ new}\ C(\bar{\mathtt{v}}) : \mathtt{T}|C$. If $e \longrightarrow^* \mathtt{throw\ new}\ C(\bar{\mathtt{v}})$, then $C \subseteq: \bar{\tau}$.

**Lemma 11 ($\Longrightarrow_c$ Exceptions).** *If $\vdash e : \mathtt{T}|\emptyset$, and $e|\emptyset \Longrightarrow_c^* e'|\mathcal{B}'$, then $\vdash e' : T'|\emptyset$.*

*Proof.* From lemma 10, we know that evaluation of $e$ through the $\longrightarrow_c$ relation cannot yield an exception value (there is no class $C$ such that $C \subseteq: \emptyset$).

During the program's evaluation, new expressions may be created through application of evaluation rule $E_c$-Spn (spawn) followed by evaluation rule $E_c$-$\eta_\emptyset$Run (execution of a null event), but these expressions will be of the form $(\mathtt{new}\ C(\bar{\mathtt{v}})).\mathtt{run}(\eta_0)$, where $C <: \mathtt{Task}$. Since the `run` method of $C$ does not declare any thrown exceptions, then, by type rule T-11, neither does the `run` method of class $C$. By lemma 10, we know that such expressions will not evaluate to an uncaught exception.

Now, we can state the main result for this section: a well-typed CTJ program either terminates to a non-exception value, diverges, or stops due to a runtime cast error.

**Theorem 5 (Soundness).** *If $P_c = \overline{CL}\ \mathtt{return}\ e$ is a CTJ program and $\vdash^c P_c$ OK, then one of the following must be true:*

– *$e|\emptyset \Longrightarrow_c^* v|\emptyset$, where $v$ is a non-exception value.*
– *$e|\emptyset \uparrow$*
– *$e|\emptyset \Longrightarrow_c^* E[(T)v]|\mathcal{B}$, where the type of $v$ is not a subtype of $T$.*

*Proof.* By induction over evaluation steps. The initial state of $P$ is $e|\emptyset$. By theorem 4, either $e$ is a value, $e$ is a runtime cast error, or there exists an $e'|\mathcal{B}'$ such that $e|\emptyset \Longrightarrow_c e'|\mathcal{B}'$. By theorem 2, $e'$ is well-typed and $\mathcal{B}'$ OK.

Next, we look at an arbitrary state $e_i|\mathcal{B}_i$ such that $e|\emptyset \Longrightarrow_c^* e_i|\mathcal{B}_i$. By the inductive hypothesis, $e_i|\mathcal{B}_i$ is well-typed. By theorem 4, either $e_i$ is a value, $e_i$ is a runtime cast error, or there exists an $e_{i+1}|\mathcal{B}_{i+1}$ such that $e_i|\mathcal{B}_i \Longrightarrow_c e_{i+1}|\mathcal{B}_{i+1}$. By theorem 2, $e_{i+1}$ is well-typed and $\mathcal{B}_{i+1}$ OK.

Thus, the program will either step forever, terminate, or become stuck due to a runtime cast error. If the program terminates, by lemma 11, it cannot terminate due to an uncaught exception.

**No lost exceptions** Note that a CTJ program never evaluates to an uncaught exception. The type system statically ensures that both the initialization expression and the `run` methods of any spawned tasks catch all exceptions thrown during evaluation. We state this more formally as follows:

**Corollary 1 (No lost Exceptions).** *If* $\vdash e : \mathtt{T}|\emptyset$*, and* $e|\emptyset \Longrightarrow_c^* e'|\mathcal{B}'$*, then* $\vdash e' : T'|\emptyset$*.*

**No lost continuations** As discussed in the introduction, when one writes event-driven programs in a continuation passing style, it is possible to drop a continuation and thus never pass the result of an asynchronous operation to its logical caller. This problem is easily avoided in CTJ programs by using asynchronous methods and `wait` calls instead of continuations. Such calls are evaluated in the same manner as standard method calls. More specifically, the language semantics ensure that, if program execution reaches an asynchronous method call or `wait` call, either evaluation of the calling expression is eventually resumed (with the results of the call), execution stops due to a runtime cast error, or the program diverges. More formally, we can state:

**Corollary 2 (No lost continuations).** *For any program state* $E[e_0]|\mathcal{B}$*, where* $e_0$ *is an asynchronous method call or a* `wait` *call, either:*

- $E[e_0]|\mathcal{B} \Longrightarrow_c^* E[v]|\mathcal{B}'$*, where* $v$ *is a value,*
- $E[e_0]|\mathcal{B} \uparrow$*, or*
- $E[e_0]|\mathcal{B} \Longrightarrow_c^* E'[(T)v]|\mathcal{B}'$*, where the type of* $v$ *is not a subtype of* $T$*.*

We first prove for the more general case where $e_0$ is an arbitrary expression. We do this using two lemmas:

**Lemma 12 ($\longrightarrow_c$ Evaluation to normal forms).** *An expression* $e$ *either evaluates to a normal form or diverges.*

*Proof Idea* This can be proven by induction over evaluation steps using theorem 3 (at each step, either an evaluation rule can be applied or a normal form has been reached).

**Lemma 13 ($\Longrightarrow_c$ Evaluation to normal forms).** *For any program state* $E[e_0]|\mathcal{B}$*, either:*

- $E[e_0]|\mathcal{B} \Longrightarrow_c^* E[v]|\mathcal{B}'$*, where* $v$ *is a value,*
- $E[e_0]|\mathcal{B} \uparrow$*, or*
- $E[e_0]|\mathcal{B} \Longrightarrow_c^* E'[(T)v]|\mathcal{B}'$*, where the type of* $v$ *is not a subtype of* $T$*.*

*Proof.* By induction over evaluation steps. By lemma 12, either $e_0 \longrightarrow_c^* e_0'$, where $e_0'$ is a normal form, or evaluation diverges. We look at each possible outcome:

- If the evaluation diverges, the second case of the lemma is satisfied.
- If the normal form is a runtime cast error, the third case of the lemma is satisfied.

– If the normal form is a `spawn` call, evaluation rule $E_c$-Spn replaces the call with an expression of the form `new` $C(\bar{\mathtt{v}})$. Then, either the entire expression is a value, satisfying the first case of the lemma, or by the inductive hypothesis, further evaluation either diverges, reaches a value, or reaches a runtime cast error.
– If the normal form is a `wait` call, evaluation rule $E_c$-Wait can be applied to add the current evaluation context $E_w[]$ and the set of waited-for events $s_w$ to the blocked set $\mathcal{B}_w$, resulting in a new program state `new Object`$|\mathcal{B} \cup (s_w, E_w[])$. The only evaluation rules that may be applied to this new state are $E_c$-Run and $E_c$-$\eta_\emptyset$Run, which select a blocked evaluation context for execution. If $\mathcal{B} = \emptyset$, then $(s_w, E_w[])$ must be selected. Otherwise, induction over evaluation steps and theorem 4 can be used to show that either $(s_w, E_w[])$ is selected for evaluation through rule $E_c$-Run, evaluation diverges, or a runtime cast error occurs. The second two cases satisfy this lemma.
  If $(s_w, E_w[])$ is selected by rule $E_c$-Run, then an event $\eta$ is selected from $s_w$ and $E_w[\eta]$ is evaluated. Either the entire expression is a value, satisfying the first case of the lemma, or, by the inductive hypothesis, further evaluation either diverges, reaches a value, or reaches a runtime cast error.

*Proof (of corollary 2).* Immediate from lemma 13.

## 5 Translating CoreTaskJava to EventJava

A CTJ program is translated to EJ by rewriting tasks and asynchronous methods to use a continuation-passing style. We describe this translation using a set of syntax transformation rules. Informally, these rules:

- Change each asynchronous method to forward its result to a continuation object passed in as an input parameter, rather than returning the result to its caller.
- Methods containing asynchronous method calls are split at the first asynchronous call. The evaluation context surrounding the call is moved to the `run` method of a new continuation class. An instance of this class is added to the parameter list of the call. The continuation class itself may need to be split as well, if the new `run` method contains asynchronous calls.
- Methods containing `wait` calls are split in the same manner. The evaluation context surrounding the call is moved to the `run` method of a new continuation class. The `wait` call is replaced by a `reg` call, with the continuation class passed as the callback parameter. As above, the continuation class itself may need to be split as well, if the new `run` method contains asynchronous calls.
- If the original body of an asynchronous method may throw exceptions to its caller, the method is changed to catch these exceptions and pass them to a special `error` method on the continuation object. This `error` method contains the same body as the `run` method. However, it replaces the use of `retVal` (the result of a call) with a `throw` of the exception.

**Translation rules** For the formalization of the translation, we consider a dynamic translation strategy where a CoreTaskJava program is translated as it executes. The original CTJ program is run until it reaches an asynchronous call or normal form. Then, the expression in the current evaluation context is translated to EventJava. If the expression is an asynchronous method or `wait` call, the class table is augmented with a new callback class which contains the evaluation context as its body. In either case, evaluation then continues until another asynchronous method or normal form is reached, upon which another translation is performed. We will show that the executions of the original TaskJava program and the dynamically translated EventJava program are observationally equivalent.

**Definition 1 ($\hookrightarrow$ relation).** *We use the symbol $\hookrightarrow$ to represent the evaluation relation created by this interleaving of execution and translation.*

We use this approach to simplify the state equivalence relation and to avoid creating extra classes to store partial results. Of course, the TaskJava compiler implementation performs a static translation. There are also slight differences in the translation strategy due to limitations in the core calculus (e.g., lack of a `switch` statement).

We assume that the translation of a CTJ program occurs after a typechecking pass, so that all expressions are elaborated with their types. We only show type elaborations where they are pertinent to the rules. In particular, they are used for distinguishing asynchronous from standard methods and for properly casting the result of an asynchronous call.

The translation relation $\rightsquigarrow$ is defined in Figure 15. Rules TR-AC1 and TR-AC2 perform a dynamic translation of asynchronous method calls. They rewrite the asynchronous method about to be called, adding a continuation parameter to which the result of the call is forwarded. In addition, if the original call may throw exceptions, these are caught in the rewritten method and passed to the continuation's `error` method. Lastly, the evaluation context of the call is moved to a newly created callback class, $C_{cb}$, based on the code templates listed in figure 16.

Rule TR-Wt translates `wait` calls to `reg` calls, moving the evaluation context to a new continuation class, which is then passed as a callback to `reg`. Rule TR-Sp translates a `spawn` call to a `reg` call with an empty event set. The last three rules handle normal forms that do not need to be translated.

*Example 1.* Figure 17 shows a simple CTJ program which demonstrates creation of a task, calls between asynchronous methods, and `wait` calls. Figure 18 shows the evaluation of this program alongside the evaluation of the dynamically translated program. Figure 19 shows the new classes and methods created by the translation.

Each evaluation step is labeled with a number. A $\Longrightarrow$ between steps in the table indicates an observable action, a $\longrightarrow$ between steps indicates a non-observable action, and a $\rightsquigarrow$ between steps indicates a dynamic translation. When

$\boxed{e_c \| \overline{CL} \rightsquigarrow e_e \| \overline{CL}'}$

$$\frac{\begin{array}{c} e_c = E[(v_0 : C_{rcv}).m(\bar{\mathtt{v}}_a)] \\ \bar{\tau} \neq \emptyset \qquad CL_{rcv} = \mathtt{class}\ C_{rcv}\ \mathtt{extends}\ C'\ \{\bar{\mathtt{f}}\ \mathtt{K}\ \bar{\mathtt{M}}\} \\ \mathtt{async}\ T_r\ m(\bar{\mathtt{T}}_a\ \bar{\mathtt{x}})\ \mathtt{throws}\ \bar{\tau}\ \{\mathtt{return}\ e_b;\} \in \bar{\mathtt{M}} \\ e'_b = \mathtt{try}\ \{\mathtt{cb.run}(e_b);\}\ \mathtt{catch}\ (\mathtt{Exception}\ e)\ \{\mathtt{cb.error}(e);\} \\ \bar{\mathtt{M}}' = \bar{\mathtt{M}} \cup \mathtt{Object}\ m'(\bar{\mathtt{T}}_a\ \bar{\mathtt{x}}, \mathtt{Callback\ cb})\ \{\mathtt{return}\ e'_b;\} \\ CL'_{rcv} = \mathtt{class}\ C_{rcv}\ \mathtt{extends}\ C'\ \{\bar{\mathtt{f}}\ \mathtt{K}\ \bar{\mathtt{M}}'\} \qquad \mathit{fresh}\ C_{cb} \end{array}}{e_c \| \overline{CL} \cup CL_{rcv} \rightsquigarrow v_0.m'(\bar{\mathtt{v}}_a, \mathtt{new}\ C_{cb}()) \| \overline{CL} \cup CL_{rcv} \cup \mathit{Callback}_{exc}(C_{cb}, E[], T_r)}\ \text{(TR-AC1)}$$

$$\frac{\begin{array}{c} e_c = E[(v_0 : C_{rcv}).m(\bar{\mathtt{v}}_a) \qquad CL_{rcv} = \mathtt{class}\ C_{rcv}\ \mathtt{extends}\ C'\ \{\bar{\mathtt{f}}\ \mathtt{K}\ \bar{\mathtt{M}}\} \\ \mathtt{async}\ T_r\ m(\bar{\mathtt{T}}_a\ \bar{\mathtt{x}})\ \{\mathtt{return}\ e_b;\} \in \bar{\mathtt{M}} \\ \bar{\mathtt{M}}' = \bar{\mathtt{M}} \cup \mathtt{Object}\ m'(\bar{\mathtt{T}}_a\ \bar{\mathtt{x}}, \mathtt{Callback\ cb})\ \{\mathtt{return}\ \mathtt{cb.run}(e_b);\} \\ CL'_{rcv} = \mathtt{class}\ C_{rcv}\ \mathtt{extends}\ C'\ \{\bar{\mathtt{f}}\ \mathtt{K}\ \bar{\mathtt{M}}'\} \qquad \mathit{fresh}\ C_{cb} \end{array}}{e_c \| \overline{CL} \cup CL_{rcv} \rightsquigarrow v_0.m'(\bar{\mathtt{v}}_a, \mathtt{new}\ C_{cb}()) \| \overline{CL} \cup CL_{rcv} \cup \mathit{Callback}_{noexc}(C_{cb}, E[], T_r)}\ \text{(TR-AC2)}$$

$$\frac{e_c = E[\mathtt{wait}\ \{\bar{\mathtt{v}}\}] \qquad \mathit{fresh}\ C_{cb} \qquad e_{cb} = \mathtt{new}\ C_{cb}()}{e_c \| \overline{CL} \rightsquigarrow \mathtt{reg}\ \{\bar{\mathtt{v}}\}, e_{cb} \| \overline{CL} \cup \mathit{Callback}_{noexc}(C_{cb}, E[], \mathtt{Event})}\ \text{(TR-Wt)} \qquad \frac{E[\mathtt{spawn}\ C(\bar{\mathtt{v}})] \| \overline{CL} \rightsquigarrow E[\mathtt{reg}\ \emptyset, \mathtt{new}\ C(\bar{\mathtt{v}})] \| \overline{CL}}{}\ \text{(TR-Sp)}$$

$$\frac{v \| \overline{CL} \rightsquigarrow}{v \| \overline{CL}}\ \text{(TR-Val)} \qquad \frac{\mathtt{throw\ new}\ C_e(\bar{\mathtt{v}}) \| \overline{CL} \rightsquigarrow}{\mathtt{throw\ new}\ C_e(\bar{\mathtt{v}}) \| \overline{CL}}\ \text{(TR-Th)} \qquad \frac{\vdash v : \mathtt{T}' \qquad \mathtt{T}' \not<: \mathtt{T}}{E[(\mathtt{T})v] \| \overline{CL} \rightsquigarrow E[(\mathtt{T})v] \| \overline{CL}}\ \text{(TR-Cst)}$$

**Fig. 15.** Translation rules

a method is called, we show both the pre-substitution and post-substitution values of each expression. The equivalence of these expressions is indicated by the $\equiv$ symbol.

The specific steps taken by each program are:

1. The program starts by running the initialization expression, `spawn T1()`, with an empty set of blocked tasks. Since the expression is a normal form, translation rule TR-Sp is invoked to convert the `spawn` to a `reg`.
2. An observable evaluation step is taken by both programs, adding the task to the blocked set of the CTJ program and the event set of the EJ program. The spawn/reg expression is replaced by a dummy value.
3. Another observable evaluation step is taken by both programs, selecting the null event $\eta_0$. In both programs, the `run` method of the task `T1` is called.
4. A non-observable evaluation step is taken by both programs, replacing the call of `run` with the method's body. Since the next evaluation step is an asynchronous call, the EJ expression is translated by rule TR-AC2. This rule creates method

$Callback_{exc}(C_{cb}, E[], \texttt{T}) \equiv$

```
class C_cb extends Callback {
  Object run(Object retVal) {
    return E[(T)retVal];
  }
  Object error(Exception exc) {
    return E[throw exc];
  }
}
```

$Callback_{noexc}(C_{cb}, E[], \texttt{T}) \equiv$

```
class C_cb extends Callback {
  Object run(Object retVal) {
    return E[(T)retVal];
  }
}
```

**Fig. 16.** Code templates used by the transformation

```
class T1 extends Task {
  Object run(Event e) {
    return this.m1(READ, SEND);
  }
  async Event m1(Event e1, Event e2) {
    return this.m2(e1, e2);
  }
  async Event m2(Event e1, Event e2) {
    return wait {e1, e2};
  }
}
spawn T1();
```

**Fig. 17.** An example CTJ program

| Step | CoreTaskjava Program State | EventJava Program State |
|---|---|---|
| 1 | spawn T1()\|∅<br>$\stackrel{\emptyset}{\Longrightarrow}$ | spawn T1()\|∅ $\rightsquigarrow$<br>reg ∅, new T1()\|∅<br>$\stackrel{\emptyset}{\Longrightarrow}$ |
| 2 | new Object()\|(∅, (new T1()).run([]))<br>$\stackrel{\eta_0}{\Longrightarrow}$ | new Object()\|(∅, new T1())<br>$\stackrel{\eta_0}{\Longrightarrow}$ |
| 3 | (new T1()).run($\eta_0$)\|∅ $\longrightarrow$ | (new T1()).run($\eta_0$)\|∅ $\longrightarrow$ |
| 4 | [$\eta_0$/e, (new T1())/this]<br>(this.m1(READ, SEND))\|∅<br>$\equiv$<br>(new T1()).m1(READ, SEND)\|∅<br>$\longrightarrow$ | [$\eta_0$/e, (new T1())/this]<br>(this.m1(READ, SEND))\|∅<br>$\equiv$<br>(new T1()).m1(READ, SEND)\|∅$\rightsquigarrow$<br>(new T1()).m1'(READ, SEND, new $CB_1$())\|∅ $\longrightarrow$ |
| 5 | [READ/e1,SEND/e2,(new T1())/this]<br>(this.m2(e1, e2))\|∅<br>$\equiv$<br>(new T1()).m2(READ, SEND)\|∅<br>$\longrightarrow$ | [READ/e1,SEND/e2,(new $CB_1$())/cb(new T1())/this]<br>(cb.run(this.m2(e1, e2)))\|∅<br>$\equiv$<br>(new $CB_1$()).run((new T1()).m2(READ, SEND))\|∅ $\rightsquigarrow$<br>(new T1()).m2'(READ, SEND, new $CB_2$())\|∅ $\longrightarrow$ |
| 6 | [READ/e1, SEND/e2,(new T1())/this]<br>(wait {e1, e2})\|∅<br>$\equiv$<br>wait {READ, SEND}\|∅<br>$\stackrel{\{READ,SEND\}}{\Longrightarrow}$ | [READ/e1, SEND/e2,(new $CB_2$())/cb, (new T1())/this]<br>(cb.run(wait {e1, e2}))\|∅<br>$\equiv$<br>(new $CB_2$()).run(wait {READ, SEND})\|∅$\rightsquigarrow$<br>reg {READ, SEND}, new $CB_3$()\|∅<br>$\stackrel{\{READ,SEND\}}{\Longrightarrow}$ |
| 7 | new Object()\|({READ, SEND}, [])<br>$\stackrel{SEND}{\Longrightarrow}$ | new Object()\|({READ, SEND}, new $CB_3$())<br>$\stackrel{SEND}{\Longrightarrow}$ |
| 8 | SEND\|∅ | (new $CB_3$()).run(SEND)\|∅ $\longrightarrow$ |
| 9 | | [SEND/retVal, (new $CB_2$())/cb, (new $CB_3$())/this]<br>(cb.run((Event)retVal))\|∅<br>$\equiv$<br>(new $CB_2$()).run((Event)SEND)\|∅ $\longrightarrow$ |
| 10 | | (new $CB_2$()).run(SEND)\|∅ $\longrightarrow$ |
| 11 | | [SEND/retVal, (new $CB_1$())/cb, (new $CB_2$())/this]<br>(cb.run((Event)retVal))\|∅<br>$\equiv$<br>(new $CB_1$()).run((Event)SEND)\|∅ $\longrightarrow$ |
| 12 | | [SEND/retVal,(new $CB_2$())/this](Event)retVal\|∅<br>$\equiv$<br>(Event)SEND\|∅ $\longrightarrow$ |
| 13 | | SEND\|∅ |

**Fig. 18.** Evaluation steps of example program.

```
class T1 extends Task {
  ...
  Object m1'()(Event e1, Event e2, Callback cb) {
    return cb.run(this.m2(e1, e2));
  }
  Object m2'()(Event e1, Event e2, Callback cb) {
    return cb.run(wait {e1, e2});
  }
}
class CB1 extends Callback {
  Object run(Object retVal) {
    return (Event)retVal;
  }
}
class CB2 extends Callback {
  Object run(Object retVal) {
    return (new CB1()).run((Event)retVal);
  }
}
class CB3 extends Callback {
  Object run(Object retVal) {
    return (new CB2()).run((Event)retVal);
  }
}
```

**Fig. 19.** Generated classes and methods from example

`m1'` based on method `m1`, creates the callback class $\texttt{CB}_1$, and changes the call of `m1` to a call of `m1'`.

5. An evaluation step is taken by both programs, replacing the call of `m1`/`m1'` with the method's body. The EJ expression is then translated by rule TR-AC2, which creates method `m2'`, creates the callback class $\texttt{CB}_2$, and changes the call of `m2` to a call of `m2'`.
6. An evaluation step is taken by both programs, replacing the call of `m2`/`m2'` with the method's body. The EJ expression is then translated by rule TR-Wt, which creates the callback class $\texttt{CB}_3$ and replaces the `wait` call with a `reg` call.
7. Both programs take an observable step. The CTJ program adds a pair containing the {READ, SEND} event set and an empty evaluation context to the blocked tasks set. The evaluation context is empty because the calling functions have no further computation – the selected event becomes the result of the program execution. The EJ program adds the pair ({READ, SEND}, `new` $\texttt{CB}_3$`()`) to the pending event set.
8. The scheduler selects the event SEND from the set and both programs take an observable step to consume the event. For the CTJ program, the resulting program state is a value and empty blocked task set, causing the program to terminate. For the EJ program, the result is a call to the `run` method of $\texttt{CB}_3$.
9. The EJ program takes a non-observable step, replacing the call to $\texttt{CB}_3$`.run` with its body.
10. The EJ program takes another step, removing the downcast to `Event`.
11. The EJ program takes a step, replacing the call to $\texttt{CB}_2$`.run` with its body.
12. The EJ program takes a step, removing the downcast to `Event`.
13. The EJ program takes a step, replacing the call to $\texttt{CB}_1$`.run` with its body.

14. The EJ program takes a step, removing the downcast to `Event`. This leaves the value SEND and an empty event set. Thus, the EJ program terminates with the same result as the corresponding CTJ program.

**Soundness of Translation** We now prove observational equivalence between a CTJ program and the EJ program obtained by evaluating the CTJ program under the $\hookrightarrow$ relation. As consequence of this equivalence, we prove the lost continuation property.

*Mapping Relation* First, we must establish a relationship between CTJ and EJ program states.

**Definition 2.** *We write $e_0 \longleftrightarrow e_1$ for CTJ expression $e_0$ and EJ expression $e_1$ if there exists an $e_0'$ and $e_1'$, such that $e_0'$ and $e_1'$ are in normal form, $e_0 \longrightarrow_c^* e_0'$, $e_1 \longrightarrow_e^* e_1'$, and one of the following is true:*

- $e_0' = e_1'$
- $e_0' = E_0'[(T)v]$ *and* $e_1' = E_1'[(T)v]$, *where the type of $v$ is not a subtype of $T$.*
- $e_0' = E_0'[\texttt{wait}\ s]$ *and* $e_1' = \texttt{reg}\ s,\ \texttt{new}\ CB()$, *where, for all $\eta \in s$, $E_0'[\eta] \longleftrightarrow (\texttt{new}\ CB()).\texttt{run}(\eta)$.*
- $e_0' = E_0'[\texttt{spawn}\ C(\bar{\texttt{v}})]$ *and* $e_1' = E_1'[\texttt{reg}\ \emptyset, \texttt{new}\ C(\bar{\texttt{v}})]$, *where, $E_0'[\texttt{new}\ C(\bar{\texttt{v}})] \longleftrightarrow E_1'[\texttt{new}\ C(\bar{\texttt{v}})]$.*

**Definition 3.** *We define a relation $\Longleftrightarrow$ between CoreTaskJava states and EventJava states:*

$$e_{c0}|(s_1, E_{c1}[])...(s_n, E_{cn}[]) \Longleftrightarrow e_{e0}|(s_1, \texttt{new}\ CB_1())...(s_n, \texttt{new}\ CB_n())$$

*where:*

- $e_{c0} \rightsquigarrow e_{e0}$ *and*
- *For all $\eta$ in $s_i$, $E_{ci}[\eta] \longleftrightarrow (\texttt{new}\ CB_i()).\texttt{run}(\eta)$.*

**Lemma 14 (Evaluation to normal form).** *If $e_c \rightsquigarrow e_e$, then $e_c \longleftrightarrow e_e$.*

*Proof Idea* We prove this lemma by structural induction on the forms of $e_c$ where a translation rule may be applied. For asynchronous method call rules TR-AC1 and TR-AC2, evaluation leads to either another asynchronous method call (for which we use the inductive hypothesis) or $e_e \longrightarrow^* e_e'$, where $e_e'$ is a normal form. For the second case, we show that $e_c \longrightarrow^* e_c'$, where $e_c'$ is also in normal form, and one of the following is true:

- $e_c' = e_e'$ (both evaluate to a value)
- $e_c' = E_c'[(T)v]$ and $e_e' = E_e'[(T)v]$, where the type of $v$ is not a subtype of $T$ (both encounter a bad cast)
- $e_c' = E_c'[\texttt{wait}\ s]$, $e_e' = \texttt{reg}\ s, \texttt{new}\ CB()$, where, for all $\eta \in s$, $E_c'[\eta] \longleftrightarrow (\texttt{new}\ CB()).\texttt{run}(\eta)$.

– $e'_c = E'_c[\texttt{spawn}\ C(\bar{\text{v}})]$ and $e'_e = E'_e[\texttt{reg}\ \emptyset, \texttt{new}\ C(\bar{\text{v}})]$, where, $E'_c[\texttt{new}\ C(\bar{\text{v}})] \longleftrightarrow E'_e[\texttt{new}\ C(\bar{\text{v}})]$.

For rule TR-Wt, both expressions are already in normal form. Thus, zero $\longrightarrow$ steps are required to reach normal form, $e_c = E[\texttt{wait}\ s]$, and $e_e = \texttt{reg}\ s, \texttt{new}\ CB()$. We then show that, for all $\eta \in s$, $E[\eta] \longleftrightarrow (\texttt{new}\ CB()).\texttt{run}(\eta)$.

For rule TR-Sp, both expressions are already in normal form, Thus zero $\longrightarrow$ steps are required to reach normal form, $e_c = E[\texttt{spawn}\ C(\bar{\text{v}})]$, and $e_e = E[\texttt{reg}\ \emptyset, \texttt{new}\ C(\bar{\text{v}})]$. Upon completion of the `spawn` or `reg` call, the placeholder will be replaced with the null event $\eta_0$, leading to the same resulting expression in both cases.

For the remaining rules, $e_c = e_e$, and the theorem is trivially true.

**Lemma 15 (Observable steps).** *If $e_c$ and $e_e$ are in normal form, and $e_c|\mathcal{B} \Longleftrightarrow e_e|\mathcal{E}$, then either:*

– *$e_c|\mathcal{B} \overset{l}{\Longrightarrow} e'_c|\mathcal{B}'$ and $e_e|\mathcal{E} \overset{l}{\Longrightarrow} e'_c|\mathcal{E}'$, where $e'_c|\mathcal{B}' \Longleftrightarrow e'_e|\mathcal{E}'$.*
– *Both $e_c$ and $e_e$ are of the form $E[(T)v]$, where the type of $v$ is not a subtype of $T$.*
– *Both $e_c$ and $e_e$ are values and $\mathcal{B}$ and $\mathcal{E}$ are empty.*

*Proof Idea* Case analysis on normal forms. Only the forms $E[\texttt{wait}\ s]$ and $E[\texttt{spawn}\ C(\bar{\text{v}})]$ are non-trivial. In each case, there is a corresponding $\Longrightarrow$ evaluation step for the translated expression which registers the same event set along with a callback for which the $\longleftrightarrow$ relation holds.

**Bisumulation** To relate executions of a CTJ program through the $\Longrightarrow_c$ and the $\hookrightarrow$ relations, we must precisely define equivalence. We use *stutter bisimulation*, which permits each relation to take an arbitrary number of non-observable steps (through the $\longrightarrow$ relation) before taking a matching pair of observable steps which interact with the scheduler. This is necessary because the translated program may need to take additional steps to reach the same execution state as the original program.

**Definition 4.** *A relation $r$ between program states is a **stutter bisimulation** relation if $\sigma_1\ r\ \sigma_2$ implies:*

1. *For all $\sigma_1 \overset{\epsilon}{\Longrightarrow}^* \sigma'_1 \overset{o}{\Longrightarrow} \sigma''_1$, there exists a $\sigma'_2, \sigma''_2$ such that $\sigma''_1\ r\ \sigma''_2$ and $\sigma_2 \overset{\epsilon}{\Longrightarrow}^* \sigma'_2 \overset{o}{\Longrightarrow} \sigma''_2$, where $o$ is either an* `In` *or* `Out` *label.*
2. *For all $\sigma_2 \overset{\epsilon}{\Longrightarrow}^* \sigma'_2 \overset{o}{\Longrightarrow} \sigma''_2$, there exists a $\sigma'_1, \sigma''_1$ such that $\sigma''_1\ r\ \sigma''_2$ and $\sigma_1 \overset{\epsilon}{\Longrightarrow}^* \sigma'_1 \overset{o}{\Longrightarrow} \sigma''_1$, where $o$ is either an* `In` *or* `Out` *label.*

**Theorem 6 (Bisimulation).** *The relation $\Longleftrightarrow$ is a stutter bisimulation relation.*

*Proof.* Consider an arbitrary pair of program states $e_{ci}|\mathcal{B}_i$ and $e_{ei}|\mathcal{E}_i$ from the original and translated programs where $e_{ci} \rightsquigarrow e_{ei}$. Suppose that the program states are related by the $\Longleftrightarrow$ relation. Then, by lemma 14, if $e_{ci}$ steps to a normal form, $e_{ei}$ steps to a corresponding normal form. Once at a normal form, if the original program takes an observable step, the translated program can take a step with the same observation, by lemma 15. The resulting program states satisfy the $\Longleftrightarrow$ relation.

We have shown the proof for only one direction – if the CTJ program takes a sequence of steps, the EJ program can take a corresponding sequence of steps. To show the other direction, we use lemmas 16 and 17 below, which state that at most one transition rule applies for each expression. Thus, once we have identified a corresponding sequence of evaluation steps between the two executions, we know that no other sequences are possible from the same initial states (assuming the same choice of selected events).

**Lemma 16 (Deterministic execution of CTJ programs).** *If* $e|\mathcal{B} \Longrightarrow_c e'|\mathcal{B}'$ *and* $e|\mathcal{B} \Longrightarrow_c e''|\mathcal{B}''$*, then* $e' = e''$ *and* $\mathcal{B}' = \mathcal{B}''$*.*

*Proof Idea* From theorem 4, we know that, if $e|\mathcal{B}$ is well-typed, then, either the program execution halts (due to normal termination or a runtime cast error), or a step can be taking via the $\Longrightarrow_c$ relation. A case analysis for each rule of the $\Longrightarrow_c$ relation shows that, if $e$ is well-typed, then no other rule may be applied to $e$.

**Lemma 17 (Deterministic execution of EJ programs).** *If* $e|\mathcal{E} \Longrightarrow_e e'|\mathcal{E}'$ *and* $e|\mathcal{E} \Longrightarrow_e e''|\mathcal{E}''$*, then* $e' = e''$ *and* $\mathcal{E}' = \mathcal{E}''$*.*

*Proof Idea* Same approach as used for lemma 16.

**No lost continuations** We can now state the lost continuation property for translated CTJ programs. Informally, in the dynamic translation of a CTJ program, if a callback is passed to an asynchronous method call or `reg` call, either the program diverges, gets stuck due to a runtime cast error, or the callback is eventually called.

**Theorem 7 (No lost continuations).** *Consider a CoreTaskJava program* $P_c = \overline{CL}_c$ `return` $e_0$ *such that* $\vdash^c P_c$ *OK. If* $e_0|\emptyset \hookrightarrow^* e'|\mathcal{E}'$*, where* $e'$ *has either the form* $E[(\texttt{new } C_r(\bar{\texttt{v}})).m(\bar{\texttt{v}},\ \texttt{new } C_{cb}(\bar{\texttt{v}}_{cb}))]$ *or the form* $E[\texttt{reg}(\{\bar{\texttt{v}}\},\ \texttt{new } C_{cb}(\bar{\texttt{v}}_{cb}))]$ *where* $C_{cb} <: \texttt{Callback}$*, then either* $e'|\mathcal{E}'$ *diverges or* $e'|\mathcal{E}' \hookrightarrow^* e''|\mathcal{E}''$*, where* $e''$ *has either the form* $E[(\texttt{new } C_{cb}(\bar{\texttt{v}}_{cb})).\texttt{run}(v)]$ *or the form* $E[(T)v_{err}]$*, where the type of* $v_{err}$ *is not a subtype of* $T$*.*

*Proof.* We use theorem 6 to construct a proof by contradiction. Consider an arbitrary CTJ program fragment of the form $E[e_0]$, where $e_0$ is an asynchronous call or wait call, and $E[]$ contains an observable action based on the result of this call. By corollary 2, the subexpression $e_0$ either evaluates to a value, diverges, or

reaches a runtime cast error. By translation rules TR-AC1, TR-AC2, or TR-Wt, the call will be translated to one of the EventJava forms listed in the theorem above.[3] The evaluation context containing the observable action will be moved to a callback in the translated program. If this callback is never called (violating theorem 7), the observable action in $E[]$ will not occur, violating theorem 6.

# 6 Implementation

## 6.1 Compiling TaskJava Programs to Java

The TaskJava compiler implements a source-to-source translation of TaskJava programs to (event-driven) Java programs. In this section, we describe the differences between our compiler implementation and the formalized translation presented in section 3. We refer to calls to `wait` and to `async` methods collectively as *asynchronous calls*. Note that this translation is only needed for methods containing asynchronous calls — all other methods are left unchanged.

Figure 20 shows the compiler's output for the `RequestTask` class of figure 4. We will use this as a running example in the following description of the translation approach.

**CPS transformation of Tasks.** The compiler uses continuation-passing style to break up the `run` methods of tasks into a part that is executed up to an asynchronous call and a continuation. Rather than implement the continuation as a separate class, we keep the continuation within the original method. The body of a task's `run` is now enclosed within a `switch` statement (e.g. lines 7 - 32 of `RequestTask`), with a case for the initial code leading up to the first asynchronous call and a case for each continuation. An integer field (called `_state._step`) is used to track the next continuation to run.

Callback classes are still created — these provide a standard interface for the callee to invoke upon completion of a call. In our example, the class `run_callback` (lines 40 - 56) is created as a callback for `RequestTask`. To resume a task, the callback simply sets the `_state._step` variable to the correct value, sets the result of the call (this is stored in the field `_state._retVal`), and re-invokes the task's `run` method. This approach is necessary because the Java compiler does not support a tail call optimization. In a pure continuation passing style translation (as used in our formalization), loops which cross asynchronous calls could result in a stack overflow.

**Task state.** Any state which must be kept across asynchronous calls (e.g. the next step of the switch statement) is stored in a new `_state` field of the task (line 3 in the example). An inner class is defined to include these new fields (`RequestTask_state`, lines 35 - 39).

If local variables are declared in a block which becomes broken across continuations, they must be declared in a scope accessible to both the original code

[3] We assume that the `Callback` base class is not available to CTJ programmers. Thus, subclasses of `Callback` appearing in a translated program must have been generated from one of these three translation rules.

```
01 public class RequestTask implements Task {
02   private CharChannel ch;
03   RequestTask_state _state = new RequestTask_state();
04   public RequestTask(CharChannel ch) { this.c = c; }
05   public void run() {
06     while (true) {
07       switch (_state._step) {
08         case 0:
09           _state.rdr = new TaskIO.Reader(ch);
10         case 1:
11           rdr.readLine(new run_callback(this, 2));
12           return;
13         case 2:
14           try {
15             if (_state._error!=null) throw _state._error;
16             _state.filename =
17               parseRequest((String)_state._retVal);
18             _state.sendData = readFile(_state.filename);
19             TaskIO.write(ch, _state.sendData,
20                          new run_callback(this, 3));
21             return;
22           } catch (Exception e) {
23             ch.close(); return;
24           }
25         case 3:
26           try {
27             if (_state._error!=null) throw _state._error;
28             _state._step = 1; break;
29           } catch (Exception e) {
30             ch.close(); return;
31           }
32       }
33     }
34   }
35   static class RequestTask_state extends Object {
36     String fileName; EventIO.Reader rdr;
37     CharBuffer sendData; int _step = 0;
38     Object _retVal; Object _error;
39   }
40   static class run_callback extends tj.runtime.Callback {
41     RequestTask task; int nextStep;
42     run_callback(RequestTask task, int nextStep) {
43       this.task = task; this.nextStep = nextStep;
44     }
45     public void run(Object retVal) {
46       task._state._retVal = retVal;
47       task._state._error = null;
48       task._state._step = this.nextStep;
49       task.run();
50     }
51     public void run(Exception e) {
52       task._state._retVal = null; task._state._error = e;
53       task._state._step = this.nextStep;
54       task.run();
55     }
56   }
57 }
```

**Fig. 20.** Translated code for `RequestTask`

and the continuation. Currently, we solve this problem by changing all local variables to be fields of the `_state` object.

**Asynchronous methods.** `async` methods are translated in a similar manner to tasks. However, since simultaneous calls of a given method are possible, the `_state` object is passed as a parameter to the method, rather than added as a field to the containing class. The `_state` object is created at the start of the call, stored within each callback, and passed to the method when it is resumed. The original parameters to the method call are also stored in this object.

**Loops.** If an asynchronous call occurs with a loop, the explicit loop statement (e.g. `while` or `for`) is removed and replaced with a "branch and goto" style of control flow, simulated using steps of the `switch` statement.

In our example, `case` 1 at line 10 represents the top of the original while loop. At line 28, we have reached the bottom of the loop and return back to the top by setting the current step back to 1 and then breaking out of the `switch`. The entire `switch` statement has been enclosed in a loop (line 6) to enable control to return to line 10 without a recursive call.

**Exceptions.** Exceptions are passed from callee to caller via a separate `error` method on the callback. The callback assigns the exception to the `_error` field of the `_state` object and then re-invokes the `run` method (lines 52 - 54 in the example). When a call may have thrown an exception, the continuation code checks whether the `_error` variable has been set (lines 15 and 27). If so, it re-throws the exception. The `catch` block is duplicated across each continuation that the original `catch` block enclosed.

### 6.2 The scheduler

We wish to avoid making our compiler dependent on a specific scheduler implementation and its definition of events. One approach (assumed by the examples of section 2) is to specify a scheduler to the compiler, perhaps as a command line option. The compiler then replaces `wait` calls with event registrations for this scheduler. This may still require the compiler to make some assumptions about the signature of the scheduler's `register` call.

We chose a more flexible approach in our implementation. We do not include a `wait` call at all, but instead provide a second type of asynchronous method — `asyncdirect`. From the caller's perspective, an `asyncdirect` method looks like a value-returning asynchronous method with an implicit (rather than explicit) callback.

However, the declaration of an `asyncdirect` method must contain an explicit callback. No translation of the code in the method's body is performed — it is the method's responsibility to call the callback upon completion. Typically, a direct asynchronous method registers an event, stores a mapping between the event and the callback and then returns. Upon completion of the event, the mapping is retrieved and the callback is invoked.

This approach easily permits more than one scheduler to be used within the same program. Also, existing scheduler implementations can be wrapped with `asyncdirect` methods and used by TaskJava.

## 7 Case Study

**Fizmez.** To evaluate TaskJava in the context of a real application, we modified an existing program to use interleaved computation. We chose *Fizmez* [5], a simple, open source web server, which originally processed one client request at a time. We first extended the server to interleave request processing by spawning a new task for each accepted client connection. To provide a basis for comparison, we also implemented a hand-coded, event-driven version of Fizmez that uses explicit continuation passing.

**Task version.** The structure of the task implementation is similar to the example web server of figure 4. Each iteration of the server's main loop accepts a socket and spawns a new `WsRequest` task. This task reads HTTP requests from the new socket, retrieves the requested file and writes the contents of the file to the socket.

The original Fizmez server used standard blocking sockets provided by the `java.io` package. To use Fizmez with TaskJava, we built a (reusable) asynchronous method layer on top of Java's nonblocking I/O package (`java.nio`). This layer, which is similar to the `TaskIO` example of Figure 3, provides a blocking API that subsets the standard `java.io` package, allowing us to convert I/O calls to TaskJava by only changing class names in field and method argument declarations.

Overall, we were able to maintain the same organization of the web server's code as was used in the original implementation. We only had to create one new class, excluding the I/O library. The original request processing code was part of the main web server class. We re-factored this out into a new class, `WsRequest`, since tasks are now processed concurrently, requiring per-task state.

**Explicit event version.** As with the TaskJava version, we built our hand-coded event-driven implementation on top of a reusable non-blocking I/O library (much like the `EventIO` library of figure 1).

The event-driven implementation required major changes to the original Fizmez code. The web server no longer has an explicit main loop. Instead, an accept callback re-registers itself with the scheduler to process the next connection request. More seriously, the processing of each client request, which is implemented in a single method in the original and TaskJava implementations, is split across six callback classes and a shared state class in the explicit event implementation.

**Implementation experiences.** Figure 21 demonstrates the issues we faced when refactoring Fizmez for the event-driven implementation. Part (a) shows a sequence of method calls which appear in the original and task-based implementations. This sequence sends HTTP response headers to the client.

Since each call involves writing a string to the client's channel, in the event-driven version, each call must occur in a separate continuation invocation. In addition, these calls are made from two different places — once when taking the response from a cache and once when reading the response from a file.

```
  sendCode("200");
  updateClient("Server: "+ws.getConf("ServerString")+"\n");
  updateClient("Content-Length: "+
               String.valueOf(responseLength)+"\n");
  updateClient("Content-Type: "+mimeType+"\n\n");
```

(a) Version for original and TaskJava servers

```
public static class SendRespHeadersCont
  implements WriteContinuation
{
  WsRequest req;  int step;
  GenericContinuation cont;
  SendRespHeadersCont(WsRequest req) { this.req = req; }
  public void start(GenericContinuation cont) {
    this.cont = cont; this.step = 1;
    req.sendCode("200", this);
  }
  public void writeDone() {
    switch (this.step) {
    case 1:
      this.step = 2;
      req.updateClient("Server: "+
                       req.ws.getConf("ServerString")
                       +"\n", this);
      break;
    case 2:
      this.step = 3;
      req.updateClient("Content-Length: " +
                       String.valueOf(req.responseLength)
                       +"\n", this);
      break;
    case 3:
      this.step = 4;
      req.updateClient("Content-Type: "+req.mimeType+"\n\n",
                       this);
      break;
    case 4:
      this.step = 5;
      cont.run();
      break;
    default:
      throw new RuntimeException("Unexpected step value'');
    }
  }
  public void writeError(Exception e) {
    req.closeChannel("Error: " + e.toString());
  }
}
```

(b) Version for event-based server

**Fig. 21.** Code to send HTTP response headers

| Client threads | Latency(ms) | | Throughput(req/sec) | |
|---|---|---|---|---|
| | Event | Task | Event | Task |
| 1 | 33.0 | 31.1 | 30.2 | 32.1 |
| 25 | 76.8 | 79.2 | 322.1 | 306.3 |
| 50 | 112.4 | 120.0 | 443.4 | 413.5 |
| 100 | 187.6 | 197.0 | 351.0 | 262.2 |
| 200 | 317.3 | 345.8 | 403.5 | 225.8 |
| 300 | 455.8 | 462.4 | 324.2 | 328.6 |
| 400 | 601.4 | 695.9 | 216.0 | 212.0 |

**Table 1.** Web server performance test results

One could implement this using four callback classes, where each callback in the sequence calls the next. The continuation to be called upon completion of the four writes would be passed along from callback to callback. To avoid creating so many classes, we used an alternative approach: we created a single callback and used a step number variable to keep track of which callback should be called next. Part (b) of figure 21 shows our implementation. Unfortunately, our original implementation of the `switch` logic contained a bug — we left out the `break` statement from case 3 and the assignment to `step` in case 4. This caused the continuation to be invoked twice.

### 7.1 Performance Experiments.

We compared the performance of the TaskJava and explicit event-driven web server implementations using a multi-threaded driver program that submits 25 requests per thread for a 100 kilobyte file (stored in the web server's cache). Latency is measured as the average time per request and throughput as the total number of requests divided by the total test time (not including client thread initialization).

The performance tests were run on a Dell PowerEdge 1800 with two 3.6Ghz Xeon processors and 5 GB of memory. Table 1 shows the experimental results. The columns labeled "Event" and "Task" represent results for the hand-coded event-driven server and the TaskJava server, respectively.

The overhead that TaskJava contributes to latency is within 10%, except at 400 client threads, where it reaches 16%. The throughput penalty for TaskJava is low up through 50 threads, but then becomes more significant, reaching 44% at 200 threads. Above 200 threads, the total throughput of both implementations drops, and the overhead becomes insignificant.

These results are not surprising, as the hand-coded event implementation has inherent performance advantages over the TaskJava version. We have not yet made any efforts to optimize the continuation-passing code generated by our compiler. For many applications, the readability and reliability benefits of TaskJava outweigh the downside of the performance cost. Over time, this group

of applications should grow larger, as we reduce the penalty by optimizing the code generated from our compiler.

**Potential improvements.** The most significant source of overhead in TaskJava is the allocation of callback/continuation objects. The hand-coded server implementation preallocates and reuses callbacks. For example, in the web server, the callbacks needed for each connection are allocated at the same time as the associated request object. These callbacks are then reused until the associated connection is closed. In contrast, the TaskJava compiler currently allocates a new callback for each asynchronous call.

In the general case, it is difficult for a compiler to preallocate callbacks. Preallocation requires an interprocedural optimization to determine placement of callback instances to avoid simultaneous calls using the same callback. For example, in the hand-coded implementation, we associate reused callbacks with each connection, as we know that, by design, there will be only one read or write request pending on a given connection at a time. In future work, we intend to enhance the TaskJava compiler to preallocate any callbacks used within a loop at the start of the containing method. This does not require any interprocedural analysis and should significantly reduce the allocation overhead of TaskJava.

## 8 Related Work

Event-driven programming is pervasive in many applications, including servers [18, 19] and routers [15], GUIs, sensor networks applications [12, 13], and long-running business processes [4]. In [2], event-based and thread-based styles are broken into two distinct differences: manual vs. automatic stack management and manual vs. automatic task management. Threads provide automatic stack and task management, while events provide manual stack and task management. By this classification, TaskJava provides manual task management and automatic stack management. A hybrid cooperative/pre-emptive approach to task management is also possible in TaskJava by using a thread-pooled scheduler. Asynchronous methods in TaskJava make explicit when a method may yield control, addressing the key disadvantage of automatic stack management cited by [2].

**Coroutines and cooperative threading.** Many implementations exist for coroutines or cooperative threading in C and C++ (State Threads [21] and GNU Pth [8], for example). In fact, [8] lists twenty such implementations. Like TaskJava, these frameworks avoid race conditions by only scheduling context switches when a thread makes a blocking I/O call or explicitly yields. Context switching is implemented through C or assembly-level stack manipulation. This enables one to preserve the natural control flow of the program, avoiding the lost continuation and exception problems. Unfortunately, stack manipulation is not possible for virtual machine-based languages, like Java. In addition, cooperative threading requires a contiguous stack space to be allocated per thread, which may result in a significant overhead when many threads are created. Perhaps more importantly, these approaches are tied to a specific scheduler and notion of

events. TaskJava programs can be "linked" against any scheduler which provides the (very generic) semantics of the `wait` call. For example, one might build schedulers for application-specific events such as incoming email messages or user requests to a web-based application.

**Simplifying event systems through metaprogramming.** The Tame framework [16] implements a limited form of CPS transformation through C++ templates and macros. The goal of Tame, like TaskJava, is to reuse existing event infrastructure without obscuring the program's control flow. Continuations are passed explicitly between functions. However, the code for these continuations is generated automatically and the calling function rewritten into case blocks of a switch statement, similar to the transformation performed by the TaskJava compiler. Thus, Tame programs can have the benefits of events without the software engineering challenges of an explicit continuation passing style.

By using templates and macros, Tame can be delivered as a library, rather than requiring a new compiler front-end. However, this approach does have disadvantages: the syntax of asynchronous calls is more limited, exceptions are not supported, template error messages can be cryptic, and the implementation only works against a specific event scheduler.

Lastly, Tame favors flexibility and explicit continuation management over safety. As such, it does not prevent either the lost continuation or the lost exception problems.

**Language and compiler support for events.** The C library and source-to-source compiler Capriccio [23] provides cooperative threading, implemented using stack manipulation. It avoids the memory consumption problems common to most cooperative and operating system thread implementations by using a whole-program analysis and dynamic checks to reduce the stack memory consumed by each thread. This downside of this approach is the loss of modular compilation. Capriccio also suffers from the other weaknesses of cooperative threading — difficulty implementing on top of a VM architecture and lack of scheduler flexibility.

The language nesC [12] addresses interleaved computation for embedded applications. It provide two constructs for structuring activities: the *task*, a deferred computation mechanism, and the *event*, a handler intended for interrupts and notifications. In short, nesC is an event-driven system with direct language support for writing in a continuation passing style. As such, it suffers from the lost continuation problem — there is no guarantee that the completion event will actually be called. This approach was chosen by the designers of nesC because it can be implemented with a fixed-size stack and without any dynamic memory allocation.

**Continuations.** Continuations in the Scheme programming language [1] permit a programmer to save the current stack and resume execution from a different context. TaskJava's asynchronous methods are a limited form of continuation. Although asynchronous methods do not support some programming styles possible with continuations, providing a more limited construct enables the TaskJava compiler to statically and modularly determine which calls may be saved and

later resumed. This limits the performance penalty for supporting continuations (such as storing call state on the heap) to those calls which actually use this construct.

**Static analysis of event-driven programs.** Techniques for analyzing and verifying event-driven systems has been an active research direction (e.g., [7, 11]). Hybrid approaches are also possible. For example, [6] implements a combination of library design with debugging and program understanding tools. TaskJava has the potential to greatly aid such techniques, by making the dependencies among callbacks and the event flows in the system syntactically apparent.

## 9 Conclusion

We have described the task programming model and its instantiation in the TaskJava extension to Java. Our examples illustrate how tasks provide a clean model for interleaved computation, avoiding the need of event-based systems to break control flow across blocking calls. At the same time, our compilation strategy allows tasks to be automatically translated into efficient event-driven code. We formalized a core language for TaskJava through an extension to Featherweight Java. Based on this formalization, we have proven key properties of the language, including type soundness, freedom from lost continuations/exceptions, and observational equivalence of the translation. Finally, we demonstrated the feasibility of our approach through a case study of a TaskJava web server.

### 9.1 Future work

TaskJava is the first step toward our goal of writing robust and reliable programs for large scale asynchronous systems.

**Static analysis tools.** By providing language mechanisms that make common event-driven programming idioms explicit, TaskJava programs are not only more readable to humans, but also enable program analysis tools to understand the precise semantics of event flow. We are currently developing a suite of program analysis tools that analyze safety properties of event-driven programs, leveraging the features of TaskJava. Since the flow of events in these programs is visible to the tool, we expect the analysis to be more precise when compared to tools that have to reason about event flow through function pointers and objects.

**Language extensions.** We also plan to extend the capabilities of TaskJava. For example, the Tame framework provides fork and join constructs for the parallel initiation of asynchronous calls. We are working with the Tame developers to find a formulation of these constructs that is compatible with exceptions and that provides the safety guarantees of TaskJava.

**Formal model.** Finally, we believe that our formal model may provide new insights into event-driven programs. One such insight may be taken from the relationship between CoreTaskJava and EventJava programs. A CTJ program interacts with the scheduler in two ways: through `spawn` and through `wait`.

These interactions are both translated into event registrations. In a program written directly using callbacks, making the distinction between these cases explicit yields more information about the programmer's intent. This may help with various static analyses.

## References


1. H. Abelson, G.J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.
2. A. Adya, J. Howell, M. Theimer, W.J. Bolosky, and J.R. Douceur. Cooperative task management without manual stack management. In *Proc. Usenix Tech. Conf.*, 2002.
3. A. Appel. *Compiling with continuations*. Cambridge University Press, 1991.
4. Enterprise Java Beans. http://java.sun.com/products/ejb/.
5. David Bond. Fizmez web server. http://sourceforge.net/projects/fizmezwebserver.
6. R. Cunningham and E. Kohler. Making events less slippery with EEL. In *HotOS X: Hot Topics in Operating Systems*, 2005.
7. J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about implicit invocation. In *FSE 98: Foundations of Software Engineering*, pages 209–221. ACM, 1998.
8. R. Engelschall. Portable multithreading - the signal stack trick for user-space thread creation. In *USENIX Annual Technical Conference*, June 2000.
9. Jeff Fischer, Rupak Majumdar, and Todd Millstein. Preventing lost messages in event-driven programming. Technical Report TR060001, Computer Science Department, UCLA, January 2006. http://www.cs.ucla.edu/~fischer/papers/tjproofs.html.
10. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.
11. D. Garlan, S. Khersonsky, and J.S. Kim. Model checking publish-subscribe systems. In *SPIN 2003: Software Verification*, LNCS 2648, pages 166–180. Springer, 2003.
12. D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Network Embedded Systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on programming language design and implementation (PLDI)*, pages 1–11, June 2003.
13. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS 2000: Architectural Support for Programming Languages and Operating Systems*, pages 93–104. ACM, 2000.
14. A. Igarashi, B.C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.
15. E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The Click modular router. *ACM Transactions on Computing Systems*, 18(3):263–297, 2000.
16. M. Krohn, E. Kohler, F. Kaashoek, and D. Mazieres. The Tame event-driven framework. http://www.okws.org/doku.php?id=okws:tame.
17. N. Nystrom, M.R. Clarkson, and A.C. Myers. Polyglot: An extensible compiler framework for java. In *CC 03: Compiler Construction*, LNCS 2622, pages 138–152. Springer, 2003.
18. J. K. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the 1996 USENIX Technical Conference, January 1996.

19. V.S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proc. USENIX Tech. Conf.*, pages 199–212. Usenix, 1999.
20. B. Pierce. *Types and Programming Languages*. MIT Press, 2002.
21. G. Shekhtman and M. Abbott. State threads library for internet applications. http://state-threads.sourceforge.net.
22. R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *HotOS IX: Ninth Workshop on Hot Topics in Operating Systems*, 2003.
23. R. von Behren, J. Condit, F. Zhou, G.C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Symposium on Operating Systems Principles*, pages 268–281. ACM, 2003.
24. M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable Internet services. In *SOSP 01: Symposium on Operating Systems Principles*. ACM, 2001.
25. Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 15 November 1994.